# BASIC ADVENTURE and STRATEGY GAME DESIGN for The Apple®

A step-by-step guide to designing elaborate programs in Applesoft® Basic, including complete program listings for the original adventure "Space Derelict!" and Five-Player Draw Poker

JIM MENICK

# Basic ADVENTURE and STRATEGY Game Design for the Apple

ALSO BY JIM MENICK

*Basic Adventure and Strategy*
*Game Design for the TRS-80*

NOT FOR RESALE

# Basic ADVENTURE and STRATEGY Game Design for the Apple

JIM MENICK

**Facts On File Publications**
New York, New York • Bicester, England

Basic Adventure and Strategy
Game Design for the Apple

Copyright © 1984 by Jim Menick

All rights reserved. No part of this book may be
reproduced or utilized in any form or by any means,
electronic or mechanical, including photocopying,
recording or by any information storage
and retrieval systems, without permission in writing from the Publisher.

Apple, Apple II, Apple II Plus,
Apple IIe, and Applesoft are registered
trademarks of Apple Computer Inc.

Basic Adventure and Strategy
Game Design for the Apple

**Library of Congress Cataloging in Publication Data**

Menick, Jim.
Basic adventure and strategy game design for the Apple.

Includes index.
1. Computer games. 2. Apple II (Computer)—
Programming. I. Title.
GV1469.2.M46 1984 794.8'2 83-11683
ISBN 0-87196-064-8

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

*Designed by Giorgetta Bell McRee*

# CONTENTS

**Acknowledgments** *ix*

**Introduction** *1*

**PART 1—ADVENTURE GAMES** *5*

**1. Planning Your Universe** *7*

Subjects for Adventures *8*
How to Plan Scenarios *9*
Mapping Out the Area *12*
Simple Arrays *15*
Planting the Objects to be Manipulated—String Variables *16*
The "Space Derelict!" Shopping List—CO(X) and CO$(X) in Action *18*

**2. Laying Out the Game—The Design Format** *21*

"Space Derelict!"—The Scenario *21*
The Adventure Module Explained *23*
Procedure—The Programming Begins *24*
Sweaty Palms at the Keyboard *34*

**3. Setting the Subroutines** *35*

The Introduction *35*
Housekeeping *39*

Variable Variables *42*
Concatenation *46*
Simple File Routines for Saving Games *49*
Winning and Losing *53*

**4. Conversation with the Computer** *56*

The Nature of the Beast *56*
Words and Responses *59*
The Mechanics of the Conversation Segment *63*
Moving Around *65*
Going Other Places—Two-Letter Words *72*
Three-Letter Words *78*
A Few Choice Four-Letter Words *82*
The Rest of the Chitchat *97*
Miscellaneous Business *103*

**5. Bugging and Debugging** *106*

A Recap of the Adventure Module *106*
Getting It into Workable Shape *108*
Editing Techniques *112*

**6. Tidbits and Trivia** *116*

An Elementary Apple Clock *117*
Verifying the Second Word *119*
Changing a Game in Progress *122*
Notes on a D&D Scenario *124*
Keeping Players Out! *126*
Getting More Memory for Your Money *128*
Why Graphics? *131*

**PART 2—STRATEGY GAMES** *135*

**7. The Art of Strategy** *137*

Games, Strategy, and Assorted Balogna *137*
Algorithms—Integrating Games into Programs *140*
Going All the Way—What Makes a Good Computer Strategy Game *142*
The Game of Poker *143*

**8. Planning the Program** *147*

Structured Programming? *147*
Some Basic Principles *150*
The Poker Flow *154*

**9. Setting It Up** *159*

Random, Schmandom—and How to Unschmandomize It *159*
Arrays Made Complicated *164*
Shuffling the Cards—Make-believe Speed *168*

**10. The Look of the Game** *173*

40 vs. 80 *174*
In and out the Window *174*
VTAB, HTAB *175*
Ongoing Information *179*
V&H on the March *182*
Dropping Out *184*

**11. The Meat of the Game** *188*

Meet your Opponents *189*
The Megaloop *191*
More Matches *195*
The Sneaky Hands *198*
Naming the Hands *204*
Creating the Betting Algorithms *205*
Betting the Hands—Completing the Betting Algorithm *209*
Getting the Draw Cards *215*
Drawing the Cards on the Screen *228*
The Human Gets His Cards *229*
Setting the New Patterns of Betting *231*
The Second Round of Betting *235*
Finishing Off the Game *238*

**12. Debugging the Tough Ones and Other Choice Topics** *246*

Debugging *246*
Sound and Light *250*
Other Strategy Games *256*
The Ultimate Strategy Game *259*
Conclusion *262*

**Glossary** *263*

**Index** *272*

# ACKNOWLEDGMENTS

The program listings in this book were made from photostats of the actual programs, double-struck on a trusty Epson MX-80 dot-matrix printer. The design specifications of this book necessitated some infelicitous line breaks, but balanced against this is the nice touch of new lines breaking at each colon. This was done by using the DOUBLE-TAKE utility program written by Mark Simonsen and published by Beagle Brothers. In addition to nice program dumps, DOUBLE-TAKE has numerous other facets to its incredible personality, and I cannot imagine how I ever lived without it. Highly recommended on all counts.

My friend Peter Schneider, who eats adventures for breakfast, was most helpful in finding fleas in "Space Derelict!" And I must thank Marvin, Walter, and David/Betsy, who took me in many years ago as a player in their monthly poker game and whose styles of play I have so shamelessly recreated here. Gerry, on the other hand, I brought into the poker game myself; I think he's financially ahead after all these years, so maybe he should thank *me*. But that notwithstanding, Gerry was also the editor of this book (poker buddies tend to stick together), and he must be credited with making sure I didn't get carried away by my own words.

For buying our personal computer in the first place and then watching me slowly but steadily make it my own, for advice on business programming and system design, and most of all for extreme, enduring patience as I plugged away at this book, I thank my wife, Liz Pereira. And finally, for displaying all the marvelous wisdom of a two-and-a-half-year-old as she so kindly allowed me to work on this book during what occasionally was her time, I thank my daughter, Kate.

# Basic ADVENTURE and STRATEGY Game Design for the Apple

# INTRODUCTION

It all seemed so simple at the beginning. I knew I had a lot to learn, but all those books tucked away in the box with the computer promised to teach me everything. Plus there were two disks filled with helpful programs that would show me the computer in action. So I sat down and learned my first, most important lesson: When it comes to learning about computers, you're on your own. Six months later, when I still couldn't figure out half the programs on the System Master disk, I felt like a damned idiot . . . until I found out most Apple owners were in the same predicament.

To make up for this information gap I dove into the computer literature pool as eagerly as I had attacked my manuals, and thus was eventually able to formulate Menick's Law of Computer Literature: All computer books are either too easy or too hard. Most of the books tended to fall into the latter category (and were usually abominably written in the bargain), and I felt that if I could understand these books I wouldn't have needed them in the first place. As for the easy ones, they simply recapitulated the information in the Apple manuals without addressing any grand purpose, on the assumption that if it didn't make sense the first time perhaps another run-through might make some of it stick.

There is another category of book that does have its attraction: collections of program listings. I am not inclined to sit for endless hours going blind trying to copy some third-graders' program for "Hammurabi," but it *is* interesting to see how others have solved problems that you have encountered yourself. There is probably more to be learned from studying a well-designed program than from any other approach to programming. Seeing how others have handled things, seeing the thought processes in action, is worth a hundred lectures on the theory of subroutines. It's also interesting to note problem-solving failures. Later on in this book we'll be discussing random

numbers and the fact, which you may already know, that the Apple does *not* generate random numbers, despite all the claims to the contrary in those aforementioned manuals. Some programmers are aware of this, and in one case I've seen the game designer had created the most elaborate equation imaginable in order to beat this problem and generate random numbers for a card game. It was a series of three elaborate equations. Unfortunately they did nothing to solve the randomization problem, but they certainly did look impressive, and I had to give the designer credit for creative obfuscation. I imagine he had eventually discovered his error and just let it ride. He had probably simply had it up to here with that program and washed his hands of the whole thing. I know the feeling.

While we're on the subject of other people's programs, I might as well air a pet peeve of mine: machine-language programs masquerading as Basic programs, which seem to be the mainstay of some books and magazines. I find nothing more frustrating than sitting down to a long session with POKE this and POKE that, CALL home, etc., and being expected to make heads or tails of it. And I think this is something a lot of hobbyists share with me. POKE commands are entirely unintelligible unless you know what business is taken care of at that machine address (if you don't know what I'm talking about now, don't worry; if you did know, you'd be sure to agree with me). And I have yet to see any writer make more than the most superficial stab at explaining what he is doing when he gets around to POKEing things. Granted you probably can't write a decent Basic graphics or music program without a POKE here or there, but you can write a good game without graphics or music. And since the manuals deal with POKE in only the most casual of references, I've always wondered why writers think all readers are equally adept at it. In any case, the programs in this book will be sticking to more familiar Applesoft Basic ground, with the barest minimum of POKEs which, I promise, will be fully explained.

With a few notable exceptions, the trend in professional software these days seems to be toward high-resolution graphics adventure games. I have nothing against these—some of them are very nice to look at—but graphics alone do not a good game make, and graphics are not the subject of this book. Graphics are simply the window dressing that sells more games in today's marketplace. I'm all in favor of a nice look to any program—it looks more professional if you spell all the words correctly, for instance, a practice that more professionals could benefit from—but I'm more interested in a game that plays well than a game that just looks good. And that's what this book is all about: designing good games easily and efficiently. In case you are fully committed to graphics and want to add them to your games, however, we will discuss them enough to get you pointed in the right direction.

The approach of this book is straightforward. Part One is devoted to adventure games, because they are easier to design than the strategy games that comprise Part Two. Adventures can be designed, for the most part, without many subroutines; adventures are fairly straightforward sorts of programs, and most of the subroutines are simply of a bookkeeping nature. Strategy games, on the other hand, consist of one subroutine after the other, linked together in a logical progression but nonetheless easily long enough to get lost in. Here I will try to guide you through your own jungles.

Throughout the book the direction will be from the general to the specific.

Concepts will be explained as they pertain to Basic and games in general and then will be put to work in one of the two complete game programs given in the book. In that way there's the double reinforcement of both getting a definition and actually seeing how something works in action. And I'm assuming no prior knowledge on your part, aside from the perfunctory reading of the manuals that came with the machine in the first place, but if you already know your way around, you should still be able to find some helpful new ideas here. I'll assume you know what a PRINT statement is and how it works, and not much else, but I'm not going to beat you over the head with it either. If you're not passingly acquainted with the elementary commands you should go back to the manuals one more time to refresh yourself whenever a command comes up in this book that you don't feel 100 percent sure of. In addition, in the back of the book there's an elaborately chatty Glossary of all the terms we'll be using. Among the reference manuals, the Glossary, and the text itself you should get a good, solid feel for all of the commands and be able to use them in your own programs as the need arises.

As you read this book you might be able to see other ways of doing the same things I'm doing in my programs. There is no one way to solve a computer problem, and if you feel you can improve on my designs, feel free to do so. The point of this book is to provide comprehensive, reusable structures for game programs and to throw in a few tips that are otherwise hard to come by. I make no claims that my ways are the only ways or even the absolutely best ways; my point is that they work, and they are as simple as I can make them. Beyond that, you're on your own.

# PART 1

# ADVENTURE GAMES

# 1

# PLANNING YOUR UNIVERSE

Simply put, an adventure game is a puzzle. The player either has to collect all the treasures hidden throughout the game universe or else has to manipulate successfully through the game universe to perform some specific task, such as saving the Earth from certain destruction. The player sifts through a variety of clues and situations, puts them together in the correct way, and thereby solves the puzzle. The measure of a good adventure game is its complications: the more twists and turns, the more meat for the player to chew on. A game with six rooms, for instance, is almost by definition less interesting than a game with 12 rooms. By the same token, a game with six complications is certainly less interesting than a game with 12 complications. In the long run, creating a good adventure game boils down to creating good complications.

If a game has 30 rooms and each room has a treasure in clear sight, and all the player has to do is plot his way through the rooms and collect the treasures, the only complication is finding the rooms. A simple complication you could add to this is one of inventory, limiting the player to carrying five items at one time. Now, at least, even though the treasures are easily accessible, the player has to do a lot of running around to carry everything. But there still isn't much mystery to the undertaking. The next complication might be to booby-trap some of the treasures. The unthinking player, bending down nonchalantly to pick up the diamond necklace, finds himself bitten by a cobra who pops out from under it. The player dies immediately and in his next incarnation has to figure out a way to remove the cobra before he can remove the necklace. Needless to say, somewhere in your game you have planted either a mongoose or a snake trap or something else that will do the job. Now we're really getting complicated. Add another twist, hide a good proportion of the treasures, and you're really cooking. For that matter, why

not hide some of the rooms as well? Now our player not only is limited in the amount he can carry, he also finds it difficult to map the universe and collect the treasures. Depending on how complicated you make your complications (and that is the crux of the art of adventure designing), either the player is now in for hours of enjoyment trying to figure out what your universe is all about, or else he'll breeze right through it because it was too simple, or else he'll *never* figure it out and he'll curse you forever.

There isn't much difference between a treasure-gathering adventure and a situational adventure insofar as the individual complications are concerned. In a situational adventure the player still walks around through the game universe trying to get past individual booby traps and trying to find the hidden pieces; the only real differences are that a situational adventure should be tighter thematically and that instead of picking up treasure the player moves through plateaus of experience until he finally performs the task he has set out to do.

Two popular early games by Scott Adams clearly point up the differences, and these games are highly recommended to anyone who wants to see what a good game designer can come up with. "Adventureland" is a treasure-gathering game. Thirteen treasures are scattered around; some are easy pick-in's, others are well hidden, and almost all of them require a trick to pry them loose. There is no real theme to "Adventureland," although there are minithemes to most of the treasures; for instance, there is an ax with the magic word "Bunyan" printed on it, and there is a sign that says "Paul's Place." More I cannot say without giving away too much of the fun. In contrast, the Adams adventure "Mission Impossible" has no treasures. At the beginning of the game the player is told that a saboteur is planning to blow up a nuclear reactor; the player's mission ("if you decide to accept it") is to defuse the bomb. To do this the player has to trick his way past a number of locked doors, and past those doors he must manipulate certain objects in such a way as not to blow himself up as he proceeds. The game is won when the bomb in the reactor is deactivated.

## SUBJECTS FOR ADVENTURES

The first decision you have to make in adventure designing is easy: treasure hunt or situational game. There's no dramatic difference in the programming, so it boils down mostly to which type of game you prefer playing yourself. The real problem you face at the outset is the idea for a game. It's all well and good to want to design adventures, but your adventures ought to be original and interesting. If you're into adventure gaming you've probably already realized that just about every possible subject seems to have been covered. There are pirate games, haunted-castle games, deserted-island games, amusement-park games; you name it. But "Mission Impossible," for instance, by taking a very specific subject, managed to carve out a special niche for itself, and my recommendation is that you try to carve out similar unique niches. The first adventure game I designed was based on the film *Casablanca*. The premise was simple, and a detailed knowledge of the film was not necessary. The player took the part of Rick (Humphrey Bogart). The object of the game was to find Victor Laszlo, the French Resistance man,

and get him safely to the airport. The problem the player faced was the ubiquitous Nazi presence. In addition to Victor Laszlo and his wife (Ingrid Bergman in the film), the player had to find and manipulate a gun, a passport, letters of transit, a boat that wouldn't start, and various other items and people. An avaricious beggar in possession of a well-equipped secret hiding place, for instance, was purely of my own invention and never dreamed of by the filmmakers.

Since my personal tastes are literary and cinematic, those sources tend to provide me with the most appealing game ideas. If you're of a mind to sell your games eventually, you must of course keep in mind copyright laws and steer clear of protected material (or else pay the proprietor a usage fee). Some ideas that come off the top of my head and are free of copyright hassles would be any number of stories by Edgar Allan Poe ("The Pit and the Pendulum," for instance, sounds like a great game, whatever you might put into it). Jules Verne is similarly ripe (*Around the World in 80 Days*—there's enough material there for 80 disks!). If you're really into it you might even try Shakespeare. (*The Tempest* sounds pretty good, Ariel and Caliban, shipwrecks and magic, a desert island. Or *Macbeth,* with murders and ghosts and witches). When you start thinking about it, the possibilities are endless and fascinating; all you have to do is start thinking in the right direction.

The inspiration for "Space Derelict!" is my own, although since its development I've discovered other, similar premises. My real object when I started working on it was to legitimize the fact that the player is sitting at a computer terminal. I wanted to design a game that was realistic right at the start, so I began working from the concept of player-at-computer and branched out from there. Eventually I decided that the player would be at Mission Control and that his computer would be hooked up to a robot at the other end. The player would manipulate the robot, communicating to it through his computer terminal, and the robot's responses would be communicated back to him on the monitor. The next step was coming up with a logical scenario for such a situation, and I decided that the robot would be a sort of space probe, tirelessly traveling the solar system as an extension tool for human beings. In "Space Derelict!" our robot comes upon what is apparently a derelict ship floating in space and proceeds to examine it under the guidance of Mission Control. Keeping with the concept, I tried to make the robot sound like a robot as much as possible, since I think such twists give an adventure a little kick of their own.

The main thing to keep in mind when working over game ideas is originality. The players who try your game will, most likely, be familiar with other adventure games. To keep their interest you should present them with a new universe, a new world to roam around in, an adventure different from the other adventures they have played. Your own reading, or your own favorite films, are the best places to look for inspiration as a starting point for your creativity.

## HOW TO PLAN SCENARIOS

After you've decided the general subject of your adventure, the next step is to plan it out in its entirety. The first thing to do is draw a map, which is not

surprising since this is the first thing the players will do. At the same time you also outline the design of the play; that is, you plot out the adventure as the player will go through it, trying to get a handle on just what he will come up against. At the same time you figure out all the complications and how the player will solve them. At the same time you tally up all the objects the player will be manipulating and decide where to plant them. At the same time you go crazy trying to do all these things at the same time, and you begin to realize why a good adventure is a fairly rare commodity.

The best way to handle design is first to figure out the main object of the game and then to break the game down into smaller phases. If your game is a treasure hunt with no overall thematic progression, then you're into phases right at the outset. If your game is situational, you simply backtrack from the big complication (defusing the bomb that will blow up the galaxy or whatever) back to the little ones.

After having decided in the situational game "Space Derelict!" to use the concept of a robot that encounters a derelict space ship, I had to figure out the so-what of it all. I wanted to create a truly alien environment where objects were not necessarily what they appeared to be, or were sometimes simply incomprehensible from the human (or robotic) perspective. To make it all come together I needed an ultimate objective other than just roaming around the ship. So I lit on the idea that this apparently derelict ship was actually a bomb pointed directly at the sun, set to explode at a distance of 100 million miles from it (pretty much at Earth point), taking the entire solar system along with it. The aliens responsible for this bomb were of the opinion that this galaxy wasn't big enough for both of us, and they intended to send us out of town the best way they knew how. However, the facilities for defusing the bomb are available . . . if the Earthlings can figure them out. And the player at his Mission Control terminal is responsible for directing his robot to do just that—the fate of the entire solar system is in his hands! This scenario gave me the overall structure to work from, and the individual complications I would eventually work in would lead to the ultimate solution of the major problem, defusing the bomb.

Complications are the key to a good game, so let's take a hypothetical complication and see how it works. The diamond necklace and the cobra mentioned before will serve as a good example.

Here is the scenario. There are two rooms, east and west. You begin in the west room, where there is a cobra coiled around a diamond necklace. You enter the command TAKE NECKLACE and the computer replies, THE COBRA BITES ME. I DIE ON THE SPOT. SORRY. CARE TO TRY THIS ADVENTURE AGAIN? On the next run-through you try the command KILL COBRA, and the computer replies, TELL ME HOW. At least you're still alive, but you have to come up with some way of killing or otherwise removing the cobra. So you GO EAST, and in the east room you find a collection of musical instruments, including a sitar, a drum, and a flute. You try TAKE SITAR but that doesn't ring any bells, so you LEAVE SITAR. TAKE DRUM is similarly fruitless, so you LEAVE DRUM. Now enter TAKE FLUTE and then GO WEST. Now try PLAY FLUTE. This time the computer responds, THE COBRA DANCES RHYTHMICALLY TO THE MUSIC AND THEN FALLS ASLEEP. (In a good adventure the commands CHARM COBRA and CHARM SNAKE should work similarly well.) Now

you try TAKE NECKLACE and, sure enough, it works. That is a complication, albeit a simple one.

But let's assume that there are actually three treasures in our hypothetical adventure. We now have one, the diamond necklace, but where are the other two? We've already manipulated the drum and sitar and they were not treasures, and there are no other objects around. Here's where you get sneaky. GO EAST and TAKE SITAR again. The computer responds, WHEN YOU TOUCH THE SITAR IT TURNS INTO GOLD. Try this on the drum and it too turns into gold. The trick is, of course, that the necklace is a magic necklace with a built-in Midas touch. And now we have one complete, if simple, phase. You can't get the golden instruments until you get the necklace—everything is interrelated, and in a fairly complicated way (for the sake of explanatory convenience I will ignore the fact that the flute and the snake should also be turned into gold if the necklace were working full tilt, but in an actual adventure game you would have to take those problems into consideration).

Our hypothetical adventure had two complications—charming the cobra and using the necklace for alchemy—interrelated in such a way as to be dependent on each other in a logical progression. That is what I mean when I use the term "phase." A phase is a collection of intertwined complications.

The complications of your adventures should be the aspects that the players get stumped on; the phases should be the aspects that have the player ready to throw the computer out the window in frustration. When a player figures out any given complication he should find that figuring it out is not enough. There should be a kicker in it that further complicates it—compound interest, so to speak. He'll enjoy the game much more if he really has to work for it. Throwing the computer out the window is an accepted facet of adventure playing.

In designing your game you should first lay out the complications, then layer them into phases. You might make it such that until a player successfully completes one phase he cannot progress to the next. An example of that in our snake scenario would be a third room where, for some reason, the player needs a golden sitar to survive. Or you might layer the phases so that successfully accomplishing one is not absolutely essential but makes life easier, perhaps in our snake scenario a third room where you can kill an Indian potentate, but it would have been easier to lull him to sleep with sitar music. And so on. The key is to stay one step ahead of the player. Let the player think he's figured it out, but design the adventure so that there's always one more thing before he really has it down.

Since complications are the essence of an adventure game, you are on your own in coming up with them, but I can give you a few more ideas and examples that can serve as a starting place for your own imagination. One important thing is the concept of the locked door. You shouldn't make it too easy for your player to roam through your universe, and keeping the passageways barred is the best way to inhibit movement. The easiest way to lock a door is literally to lock it and hide the key, but there are other ways of doing it as well. You can cover a passageway with boulders too heavy for the players to lift, then hide a bulldozer somewhere, and maybe even hide the key to the bulldozer. You can make a passageway so small that a player cannot get through it—until he finds a shovel to enlarge the hole. You can

have your door so high that the player first has to find a ladder or a rope. In "Space Derelict!" I have five different kinds of locked doors: Four are literal doors and one is a "conceptual" door (when a player tries to pass through this particular passageway he gets blasted—unless he "opens" the imaginary door). There are four methods of opening these varying doors: pressing buttons, using a metal clicker, telepathy, and blasting. Needless to say, the more original your style of locking and unlocking doors, the more fun for the player.

Another complication that's nice to put in is the secret room (which, in a way, is really just another version of the locked door). A secret room can be precisely that; to discover and enter it a player must perform some particular action germane to secret-room finding. You have to give a hint that such a room exists or else the player will never find it, but you needn't make it too easy. In my "Casablanca" game the player was made aware of the need of a hiding place through dialog uttered by one of the game characters. To find the hiding place the player had to bribe a beggar. As soon as the money passed hands the player found himself in a secret room behind a brick wall in a corner of the city's bazaar. Any way you want to hide your secret rooms is fine, but make sure you've left a clue somewhere so that the player can get to it eventually.

Here are some other examples of possible complications. In our snake/flute scenario, you might provide the player with a snake, a piece of wood, and a hole-puncher—now he has to figure out how to make a flute and then use it. Or have an alligator guarding a treasure and plant a feather somewhere else—TICKLE ALLIGATOR might be an interesting command (unless your player is foolhardy enough to WRESTLE ALLIGATOR, in which case more power to him). Or perhaps you have vampires to vanquish. Provide the player with a woodpile and a hatchet and let him figure out that he has to MAKE STAKE to create his vampire weapon. Or plant a werewolf in your game, then hide a gun, a bullet-making apparatus, and a silver mine. Make the player work for his victory—don't just hand him a gun loaded with silver bullets. That's what complications are all about.

## MAPPING OUT THE AREA

Once you've roughly planned your complications and you have a good idea of the flow of your game, it's time to make a map. How precisely you wish to define your universe at this stage is up to you. You can make measured maps on graph paper, or simply plot out the areas on the nearest scrap sheet. In either case, once your map is finished, *do not throw it away!* No matter how finished you think you are, no matter how many years have passed, no matter how perfectly you think your game is running—somewhere, somehow, there's going to be a bug in it, and without your map as a reference even the most clearly laid-out program will look as though it was written in Chinese after you've been away from it for a while.

Since every area marked off on your map will appear on the monitor, for the sake of simplicity every one of these areas should be considered a room. A room is any unique area that the player can roam around in, with exits to other unique areas. A room can be an actual room, of course, but it can also

be a part of another room, or a street, a cave, a treetop, a boat, or any other area of your universe the player can explore. The reason for this is simple: If you call them all "rooms" you can assign them one variable within the program. Call them all something different and you'll get lost in the bookkeeping.

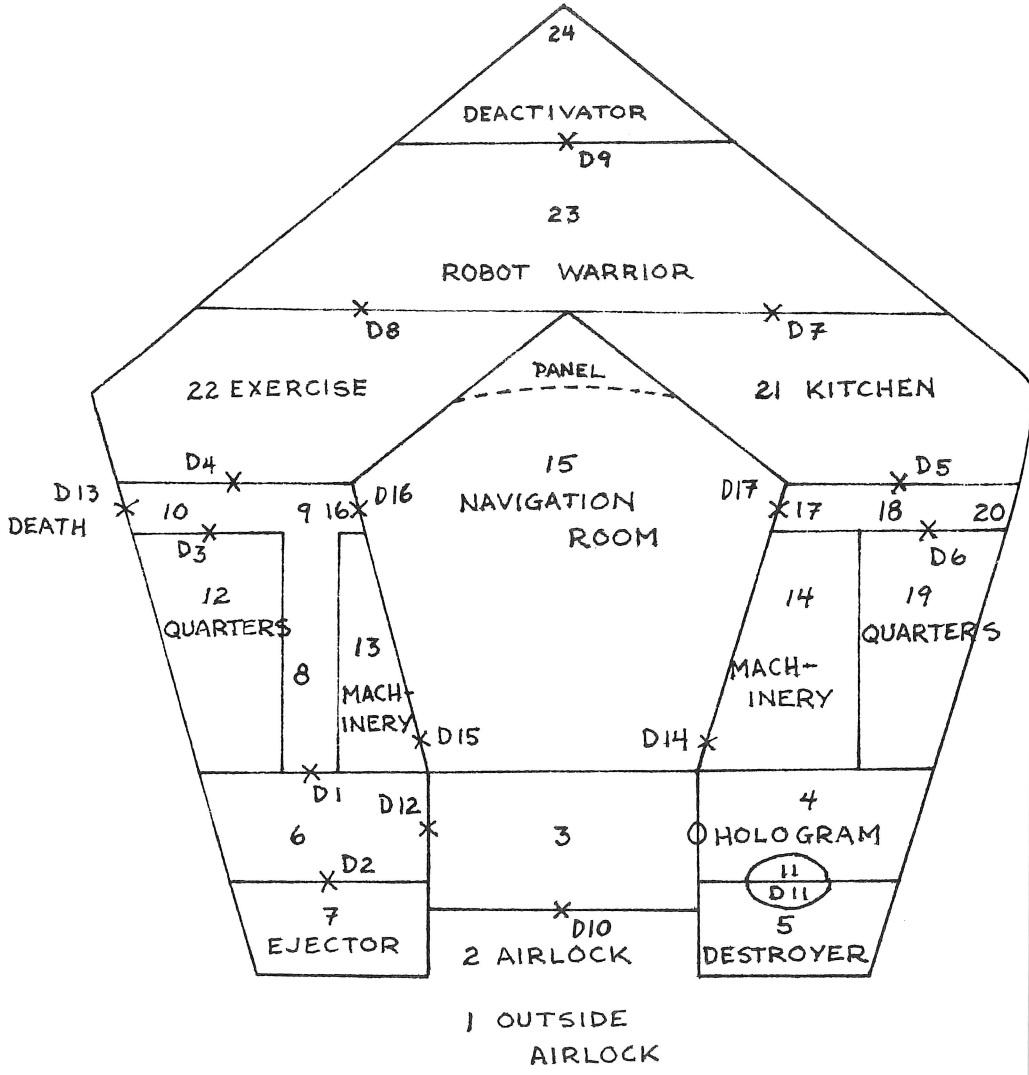
A simple map might consist of three boxes connected horizontally. Room 1 would be on the left, room 2 would be in the middle, and room 3 would be on the right. Let's say that room 1 is a barbershop, room 2 is the backroom of the barbershop, and room 3 is a parking lot behind the barbershop. Let's say further that the objects in room 1 are a barber's chair, a table with some three-year-old magazines, and some shelves with tonsorial equipment. Let's say even further that the barber's chair is a complicated space-traveling device with all sorts of buttons and whatnot—so complex, in fact, that it deserves its own room-type listing on the monitor. In that case, we'd make the chair room 4. To get from room 1 to room 4 the player would say, GO CHAIR. The monitor would change and the player would be facing a whole new area. Again, the reason for this sort of division is simplicity of variables. If you were to keep the very complicated chair as part of room 1, you would find yourself going crazy trying to keep track of it after the GO CHAIR command. This way it's another area entirely, so you can handle it the way you would handle any room. Boiling problems down to their simplest factors is one of the first laws of programming. The fewer things you have to do to obtain results, the better.

So now we have four rooms, two of which are rooms per se, one of which is a parking lot, and the other of which is a barber's chair. Draw a little box in the center of room 1 on your map to represent the chair, and now label the four rooms 1, 2, 3, and 4. If you've got enough space (and I certainly recommend this) you can also label the rooms descriptively—Barbershop, Backroom, Parking Lot, Barber's Chair. Now you've got a very clear hypothetical map of four areas with nothing in them. But if this were a real adventure you had designed, you would have numerous items with which to stock the rooms. It's best to make a list along with the map, and on that list start with room 1; identify it, then list the things that will be found there. A few descriptive lines about the room might also be helpful when it comes time to program them; it helps set the rooms in your own mind.

Now you're ready for your first real programming thingamabob, a variable called R. In our adventure programs the variable R will refer to the room number. In our example there are four possible values for R. If R = 1, then the room is the barbershop, room 1. If R = 4 then the room is the barber's chair, room 4. If R = 2392, you haven't been following closely enough, or else you've added a few extra rooms to our haircut scenario.

The map of "Space Derelict!" is reproduced on the next page. There are 24 rooms in it, so in this game R will range between 1 and 24. Some of these rooms are rooms, others are hallways, one is a door, others are parts of hallways, and one (which happens to be numbered 1) is just the starting point of the game, located in space outside the space ship.

You will see on this map various written indications of which room is which, and a few other things we'll talk about later. You'll also see a series of numbers ranging from D1 to D17. These numbers refer to doors, 17 in all. In a number of cases there are no doors between rooms, but in more


24
DEACTIVATOR
D9
23
ROBOT WARRIOR
D8
D7
22 EXERCISE
PANEL
21 KITCHEN
D4
15
D5
D13
DEATH
10
9 16
D16
NAVIGATION
ROOM
D17
17
18
20
D3
D6
12
QUARTERS
14
19
QUARTERS
8
13
MACH-
INERY
MACH-
INERY
D15
D14
D1
D12
4
6
3
HOLOGRAM
D2
11
D11
7
5
EJECTOR
D10
2 AIRLOCK
DESTROYER
1 OUTSIDE
AIRLOCK

cases than not there are. As the player goes through the game he will have to open all these doors (we'll discuss how later). The important thing for now is that we have just come upon our second variable, D, and this one I'm afraid is quite a bit more complicated than our friend R. So we'll have to break to a new section, aptly entitled:

## SIMPLE ARRAYS

Arrays are not particularly complicated, but you'd never know it by reading most explanations of them. The best way to get the hang of them (and this applies to just about every facet of programming) is to use them.

As you probably know, Applesoft Basic can handle only the first two characters of any variable. If instead of using R to denote our room variables we were to use the variable ROOM, the computer would use the letters RO and ignore the letters OM. Sometimes it's nice to use longer variables just to keep them straight in your own mind, but if in the same program with ROOM we were to use the variable ROVER, the computer would read only the RO in both and would not differentiate between ROVER and ROOM.

Arrays are a convenient way of getting around this limitation. An array is a variable that looks like this: D(14). What we're doing here is using one variable name, D, but giving it a lot of different possibilities. In fact, in "Space Derelict!" where D refers to doors, we are going to give it 17 possibilities, so our D variable can range from D(1) to D(17). D(1) is door 1, D(2) is door 2, etc. The reason we simply can't use D1, D2, . . . D17 should be obvious: Since the computer reads only the first two digits of the variable, it would treat D1 and D17 as identical, which would screw us up royally.

For the computer to allot the appropriate memory space for an array, you must first tell the computer how big your array will be. You do this by declaring the dimensions of the array. In our case now the proper dimension is 17, so we tell the computer

DIM D(17)

and now we can have 17 different doors to do with as we please.

Arrays can be a lot more complicated than that if we wish, and we'll be using more complicated ones later in our poker game—e.g., V(3,4), where the array has two parts. An array can also look like this: NW(1,0,3,4). There is a limit to the number of dimensions in an array, just as there is a limit to the number of variables your machine will hold, based on the finite number of bytes you paid for when you bought your machine. Without going into too much detail here about memory-eating variables (you can experiment with this on your own time), suffice it to say that although you don't have too much to worry about, it's something to keep in mind when you're doing your dimensioning.

The other important thing about dimensions is that almost everything the computer does begins with the value 0. So in our DIM D(17) command we have actually set aside 18 values for D, from 0 to 17. Setting it at DIM D(16) would have given us 17 values, from 0 to 16 inclusive, but in a case like this it's simpler to use tangible numbers, since we're talking about tangible doors—and I find it hard to master thinking of 0 as the first number of a

series. It takes time and space for the computer to sort out variables, so using the precise numbers including 0 can be helpful when processing speed is an issue. For example, there would be no point in our saying DIM D(30) when we're using only 17 Ds to get the job done; the computer is going to set aside those extra places even if we don't use them, so you might as well keep things as tight as possible. But ignoring that 0 factor usually won't hurt, providing there aren't too many variables to keep track of.

In "Space Derelict!" any given door is going to be either open or closed, so any given D(X) is either going to equal 0, closed, or 1, open. If D(4) = 0, then door 4 is closed. If D(4) = 1, then door 4 is open. So now we have 17 doors, each of which is either open or closed; i.e., we have DIM D(17), where D(X) = 0 or D(X) = 1.

That wasn't so bad, was it?

## PLANTING THE OBJECTS TO BE MANIPULATED— STRING VARIABLES

So far we've mapped out the rooms and marked the doors. The only major things missing from the physical structure of our universe are the movable objects lying around that the player can play with.

When I did my first adventure I gave the movable objects the variable name of CO, as in COnstant, since those objects would remain invariable throughout the game, and I've stuck with it. Actually we're talking about CO(X) because we have a number of COs lying around, and CO(X) refers to any particular one of them. In "Space Derelict!" there are 16 movable objects, or COs, so let's set it in our minds now:

DIM CO(16)

CO(1) is object 1, CO(16) is object 16, and the other numbers correspond to all the objects in between.

Unfortunately, as you can see, we now have numbers for all the objects but no names for them, so we have to flip over to the other side of the coin and become acquainted with string variables, in our particular case the string variable CO$(X).

String variables work much like numeric variables; R and D(X) were numeric variables—they were equal to numbers, R=4 or D(5)=0. The difference is that a string variable can contain anything you want it to contain. A string variable is the contents of a set of quotation marks. If we were to type into the computer the line

F$="ABRAHAM"

then ABRAHAM would be the contents of string F$. The dollar sign tells the computer that F$ is a string. If we were to type in

G$="2001"

then 2001 would be the contents of the string G$. If we now asked the computer to

PRINT F$

the computer would respond with

ABRAHAM

If we typed in

PRINT G$

we would get the result
2001
In this situation the computer doesn't care what's in the quotation marks and will simply hold those contents, be they numbers or letters or a mixture of both, and read them back to you whenever you want them.

String variables can be manipulated in a number of ways, most of which we'll discuss later. At this point it is enough to remember that a string variable looks like any variable with a dollar sign after it (e.g., F$) and that string variables conform to all the regular variable rules. We can dimension an array of string variables just as we dimension a numeric variable. In "Space Derelict!" we have 16 movable objects, each one of which has a name, and we are going to assign the string variable CO$(X) to each one. Once again we set a dimension for our array of objects, this time
DIM CO$(16)
In any adventure, the numeric CO(X) variable will correspond to the string CO$(X). Now I'll put them together to show you what I mean.

In "Space Derelict!" CO$(1) = "ODD PIECE OF METAL." After I've set this into my program, if I were to type
PRINT CO$(1)
the computer would respond with
ODD PIECE OF METAL.
However, in my game I want to know more than just *what* a particular object is; I also want to know *where* it is. That's where my corresponding CO(X) comes in. The value of CO(X) will tell me the location of CO$(X).

In "Space Derelict!" there will be two possible locations for any CO(X): Either an object is lying about in any one of the 24 rooms, or my robot is carrying the object. If my robot is carrying it, then I will assign it a value of 1. Let's assume my robot is carrying the odd piece of metal. In that case
CO(1) = 1
Regardless of which CO(X) movable object I am talking about, if that CO(X) = 1, then the robot is carrying that object. Since we've just established that CO(1) = 1, we know that the robot is carrying the odd piece of metal. Note that this logic works in both directions: If we know that the robot is carrying the odd piece of metal, then we know that CO(1) = 1.

The other possible location for a movable object is lying around in any one of the 24 rooms. To make this as simple as possible we will set 100 + R as this value for CO(X) (and believe me, that is simple—don't throw the book out the window yet). We know that R equals a room number; if R = 19, we know we're referring to room 19. So if our odd piece of metal were lying around in room 19, then CO(1) would equal 119 (100 + R, or 100 + 19). You'll see the beauty of this when we get into the program and start picking things up and dropping them off. Let's set this now:
CO(1) = 119
and play with it a minute. Our robot is in room 11, let's say, which means that R = 11. Our player enters the command TAKE ODD PIECE OF METAL. At this stage in our program the computer will know that the odd piece of metal is CO(1), so the computer will try to figure out if CO(1) = 100 + R. Since CO(1) = 119 and we're in room 11, therefore R = 11 and 100 + R = 111; so the computer will be able to respond conveniently THAT IS NOT HERE AT THIS TIME.

Or let's imagine that we are in room 19 and CO(1) = 119. In this case R = 19. This time if we try to take it, the computer reads the 100 + R, which indeed does equal 119, the value of CO(1). The program now will change the value of CO(1) to 1—remember, if we're carrying a particular CO(X), then that CO(X) = 1. How this is all done in the program you'll see later, but the fact that it can be done with these variables is the important thing now. If you're thoroughly confused, hang on a minute, because it might make more sense when you see it laid out in front of you.

## THE "SPACE DERELICT!" SHOPPING LIST—CO(X) AND CO$(X) IN ACTION

The problem with trying to explain complicated ideas like the preceding ones is that if we present examples first, they don't make sense without a previous explanation, but on the other hand if we start with the explanations they don't make much sense until you see them in action. Computers are complicated animals and even the simplest aspects, when described in writing, are going to look like gobbledygook until you let them settle into your head for a while. There are always a million things going on at once, and it takes some time to get them all straight.

In this book there are going to be two kinds of program listings. The first kind is hypothetical and will look like this:

100 GOTO JAIL:REM GO DIRECTLY TO JAIL
110 IF X = $200 THEN DO NOT PASS GO

This kind of listing will be used to explain various programming concepts relative to nothing in particular except those concepts.

The second kind of listing will look like this, the name of the program flanked by asterisks:

***Space Derelict!***

```
2260  CO$(1) = "ODD PIECE OF METAL. "
    : CO(1) = 119
2270  CO$(2) = "SMALL LEAD PENTAGON."
    : CO(2) = 104
2280  CO$(3) = " CRYSTAL PENTAGON.  "
    : CO(3) = 107
2290  CO$(4) = "UNIDENTIFIED WEAPON."
    : CO(4) = 121
2300  CO$(5) = "   PENTAGONAL BOX.  "
    : CO(5) = 105
2310  CO$(6) = "  CRYSTAL STAR.     "
    : CO(6) = 0
2320  CO$(7) = "SMALL LASER PISTOL. "
    : CO(7) = 1
2330  CO$(8) = "SMALL FLASHLIGHT.   "
    : CO(8) = 1
2340  CO$(9) = "LARGE BLADE KNIFE.  "
    : CO(9) = 121
2350  CO$(10) = "  METAL BUCKET.     "
    : CO(10) = 121
```

```
2360  CO$(11) = "RUINED ALIEN ROBOT.  "
    : CO(11) = 105
2370  CO$(12) = "STRONG HEMP ROPE.    "
    : CO(12) = 0
2380  CO$(13) = "GIANT FIVE-LEGGED ROBOT."
    : CO(13) = 123
2390  CO$(14) = "SMALL AREA OF WATER AT EDG
        E OF ICE.      "
    : CO(14) = 0
2400  CO$(15) = " FILLED BUCKET.      "
    : CO(15) = 0
2410  CO$(16) = "DEACTIVATED ROBOT."
    : CO(16) = 0
```

The difference will be in those asterisks in the heading, plus the distinctive typeface. If a listing has a heading, it will be either *Space Derelict* or *Five-Card Draw* and the listing will be part of one of these two programs. If you want to enter these programs into your machine, then type in those listings whenever they appear. There will be no uninterrupted listing of either program in the book, so to get them up you will have to enter them a piece at a time.

If you're interested in getting "Space Derelict!" up and running, you'd better get thee to a computery and start typing now! Copy the listings above for 2260–2410 EXACTLY as they appear (and pay particular attention to the spacings used within the quote marks). These lines in the "Space Derelict!" program establish the name and position of each movable object (the use of the colon allows us to put these two inextricably related variables on the same lines, side by side). This piece of program will be executed before the actual game play begins, in order to set our two variables CO(X) and CO$(X). Let's see what we have here.

Line 2260 is our odd piece of metal, and as you can see, it really does have a CO(1) value of 119. That means that the odd piece of metal is in room 19 (again, that's 100+R, where R is equal to the number of the room, in this case 19). Lines 2270 through 2300 follow a similar pattern. The small lead pentagon that is CO$(2) is located in room 4, obviously, because CO(2)=104, i.e., 100+R or 100+4. And so on. We'll skip 2310 for a minute and look at 2320. Since CO(7)=1, we know that our small laser pistol, CO$(7), is being carried by our robot. Our small flashlight is similarly in the hands of our puppet robot, since CO(8)=1. The mean-sounding giant five-legged robot of CO(13) is, of course, lying in wait in room 23. That leaves those five lines (2310 and 2370–2410, excluding 2380) where CO(X)=0. Earlier I said that there were only two possible locations for any CO(X), either in the hands of our robot or in a room, a CO(X) value of either 1 or 100+R, respectively. So shoot me, I lied a bit.

The point here is not to be misleading but to show you that in an adventure you can put an object anywhere you want simply by assigning any meaningful number to it. Any 100+R value places the object in a room, and a value of 1 places it on your puppet's person, but that leaves an awful lot of numbers for you to play around with if you are so inclined. Perhaps you have a few roving monsters in your adventure, and these monsters are capable of picking up objects. Let's say there are 10 monsters. If any of them were to be in

possession of a CO$(X), then you could set it in your program as:
CO(X)=200+M:REM 200 INDICATES CO$(X) IN HANDS OF MONSTER, M INDICATES WHICH ONE
Or say you want to add locations to your adventure more specific than the rooms. You might have room 14 in which there is a grand piano, and you want to be able to hide a celadon croissant chopper, CO$(22), in that piano. You could assign, say, a value of 2 to the piano, and therefore CO(22)=2 would put that strange little piece of kitchen equipment right where you want it. The thing to remember is that the values of CO(X) can be flexible. I've assigned the number 1 to items of inventory (i.e., items on the person of my puppet robot), and 100+R to objects in rooms. You can add as many variations to this as you wish.

All of which doesn't explain my use of CO(X)=0. The value 0 is very specific to these individual items; in essence these items do not exist in the game until certain other criteria are fulfilled. The crystal star of 2310 is well hidden, as is the rope of 2370. When the player finds them, their values will be changed to the appropriate 100+R. CO$(14) is water that will exist only if the player is clever enough to melt some ice, and the filled bucket of 2400 will at some point replace the metal bucket of 2350, as soon as the player thinks to fill CO$(10) with the water he has created at CO$(14). These 0-value CO(X)'s are reserves lurking around, to be called up to action when the time is right. Don't forget that concept of complications. If we were to leave everything in plain sight it would be just too easy; in "Space Derelict!" my 0 value of CO(X) is my catchall hiding place for some of my more devious complications.

Now, having entered lines 2260–2410 into your computer, you can save "Space Derelict!" and turn the machine off. You now have a complete list of the manipulable objects in the game—the "shopping list," if you will—and you also have your first listings on your way to assembling the entire game. You are on your way and loaded for bear.

# 2

# LAYING OUT THE GAME- THE DESIGN FORMAT

## "SPACE DERELICT!"—THE SCENARIO

You know a little bit about "Space Derelict!" already; now I'll fill in the blanks and outline the play of the game. The level here is pretty basic, with not that many rooms or manipulable items. It was designed primarily as an instructional tool, and as such I've tried to keep the complications manageable. But the nature of an adventure remains the same regardless of its complexity. In fact, "Space Derelict!" would make as good an introductory game for players as it does for game designers. So let's run through it and see how it goes.

After an introduction explaining how the game is played and setting up the player's role as Mission Controller, our puppet robot arrives at the UFO (room 1 on the map on page 14). There is nothing to be seen but an airlock ahead of him. When he enters the airlock (room 2) he is faced with a locked door and a panel of three different-colored buttons. If he presses the right button, the door opens; if he presses the wrong button, he either shuts a door behind him or else is blasted into oblivion. Assuming he has pressed the right button, he now enters the ship (room 3).

Once inside the ship our robot encounters a western door that cannot be opened. He has no alternative but to travel east (to room 4), where he will find a small lead pentagon, a hologram, and a strange device. The hologram proves unintelligible, but as he approaches the strange device a message is broadcast from the inhabitants of the planet Krenn. Our friends the Krennians now tell us that the ship is a bomb that will blow up the solar system; it can be defused, but it is doubtful that we'll be able to do it successfully. End transmission. Needless to say, our robot picks up the small lead pentagon; this turns out to be a clicking device that will open some of the doors.

Room 11 to the south is a passageway into room 5. If the robot looks into room 5 from room 11 he can see the debris of a destroyed robot. If he enters room 5 he is blasted into oblivion. At this point there is no way he can enter that room and survive to brag about it.

Our robot now heads back to room 3. By clicking the pentagon he opens the previously locked western door and gains access to empty room 6. Another click opens the southern door (but not the northern one). Entering the southern room he finds a crystal pentagon and another panel of buttons. These buttons serve no earthly purpose but to cause mischief; pressing them invites another blast into oblivion.

This crystal pentagon is an interesting little gizmo. Our robot recognizes it as organic, and eventually our player must discover that it is a telepathic door opener. If the robot thinks (or says) "OPEN", then a door will open. A sneaky trick—too sneaky?—and it will take the player a while to figure it out. This telepathic door opener also allows entrance into room 5 back in the other direction, where the player will find a pentagonal box, quite empty. We are now ready to progress into the bowels of the ship.

The hallway (room 8) leads to the crossroads (room 9). The robot can follow a set of footprints that lead to a western door (and another oblivion blast). The room south of corridor 10 is the crew's quarters, now empty. The door at area 16 opens into the control center of the ship (room 15). Here our robot finds a locked panel of controls and a crystal star. Unfortunately, the crystal star won't budge when he tries to remove it.

Two southern doors open into machinery rooms (13 and 14), neither of which yields any information. Exiting northeast brings our robot to corridor 17–18–20. Going south from 18 is another crew's quarters, where our robot can pick up the odd piece of metal we talked about in the previous chapter. North of 18 is the kitchen, where our robot can obtain a metal bucket and a knife.

If our robot goes north from here into room 23 it's curtains, so let's backtrack through the main control room first. Using the odd piece of metal, our robot can now remove the crystal star. Down the corridor and up to room 22 is some sort of exercise area with climbing ropes hanging down, a swimming pool totally frozen (it's cold in outer space), and even a marked-off court for some kind of ball game. Climbing the ropes is fruitless, but if our robot is so directed he can cut the rope and use it for other purposes (all of which will prove futile—the rope is a red herring). Melting some of the ice and putting it in the bucket will be a good idea, however. Now if we go north to room 23 we find that a robot is waiting to dish out the traditional oblivion blast. This robot is almost omnipotent. As it turns out, the only thing that will put him out of commission is a bucket of water over the head. We can now proceed up to room 24.

Unfortunately, neither our clicker nor our telepathic device opens this final door, but a good blast from the laser pistol does the trick. Once inside we find a panel with a slot in it. This is the bomb defuser. First we must insert the crystal star into the pentagonal box (clues have been given as to the advisability of this). Then we insert the filled box into the slot. And a message from the Krenn congratulates us on our cleverness but warns us that we will hear from them again. End of game.

It doesn't look all that complicated laid out like this (as I said, this game

was specifically written as an instructional tool), but take my word for it—nobody's going to walk through this baby in five minutes. The biggest pain in the patoot will be the offhanded blasts into oblivion; the player would be well advised to save the game whenever he's trying any new maneuvers. Of course, now that you've read the description *you* could walk through it in one sitting, but your situation is different: You're learning how to create other adventure games. Let's go on now to the actual program format, 100 percent guaranteed for any adventure you set your mind to. A simpler setup you couldn't imagine.

## THE ADVENTURE MODULE EXPLAINED

When you get right down to it, there isn't very much going on in an adventure at any one time. The player is in a room; there may or may not be other objects in the room with him. And the player may or may not be carrying certain items. That's all there is to it. Most of adventure game module is concerned with one aspect or another of this basic situation and boils down mostly to moving the player and/or objects from one room to another.

The format of the basic module breaks down like this:

Introduction/housekeeping routines
Room layout
Room objects subroutine
Inventory subroutine
Game-saving routines
Miscellaneous-business routines
Winning and losing scenarios
Conversation with the computer

We'll take these one at a time and talk about what they do. Keep in mind that they are valid for any adventure, not just "Space Derelict!"

**Introduction/housekeeping routines.** The introduction is just that. It includes such things as the game title and author and if necessary some instructions on how to play the game. Further, you might need to set up certain plot elements to give the player some background. This is where all that would take place.

By housekeeping I mean setting variables (also known as initialization). The housekeeping routine would include, for instance, the list of CO(X) and CO$(X) values we talked about earlier plus a bunch of other variables we'll talk about later. This subroutine in "Space Derelict!" takes up only 48 lines, and even an adventure five times more difficult wouldn't require that much more housekeeping space. It's this sort of simplicity that allows the game designer to concentrate on the play of the game rather than data crunching.

**Room layout.** The room layouts are the physical descriptions of the rooms that will appear on the video monitor. One rather long routine does all the work of setting and resetting the text window, figuring out what items must be displayed as being in the particular room, and then doing the actual

display. With 24 rooms in "Space Derelict!," the room layout routine is 125 lines long. Judicious editing once you get the hang of it, plus an alternate method presented in Chapter 6, could make it even smaller.

**Room objects subroutine.** An offshoot of the room layouts, this 5-line piece of business reads the CO(X) values of all the movable objects and decides whether each object belongs in the room at hand.

**Inventory subroutine.** This is a six-line piece of business much like the room objects subroutine. This time the CO(X) values are read to determine whether a particular item is on the person of the puppet robot—i.e., whether the various CO(X) values equal 1.

**Game-saving routines.** These are exactly that. One of these routines allows you to save a game in progress; the other allows you to pick up a game where you left off. In "Space Derelict!" these routines total about 40 lines; their length depends on the number of variables in a given adventure. Again, it's hard to imagine a reasonable adventure with very many more variables than we have here.

**Miscellaneous-business routines.** The less miscellaneous business in your adventure the better. It's best if you can contain the game within the confines of the major routines. But sometimes this just isn't possible and a miscellaneous area in your program becomes a necessity. In "Space Derelict!" I use this area to monitor the attacks of the five-legged robot in room 23. The problem with miscellaneous areas is that it can be hard to get them to work right, which is why they are best avoided.

**Winning and losing scenarios.** You need some place to go in your program if the player wins. Also, if you've built in any setups where a player could get killed, you need a resolution for that, too. That's what these are all about.

**Conversation with the computer.** Well over 500 lines of "Space Derelict!" are devoted to the player's keyboard conversation with the computer, and any expansion of this number would result in a proportional improvement in the adventure. The more conversation the computer is capable of, the better the game. In this adventure the player enters either one- or two-word commands, and the computer responds accordingly. If the program recognizes the command, an appropriate action is taken, and the game progresses. If the program only partially recognizes the command, the player is advised that he's warm but not quite on the money. If the command is a complete bust, the player is told to try something else. In "Space Derelict!" the program recognizes about 50 verbs and well over 100 nouns; more importantly, it knows what to do when it hears them.

## PROCEDURE—THE PROGRAMMING BEGINS

You now have designed your adventure in all its complications, and you're ready to put it on a disk to see how it runs. For almost any program other

than an adventure it would be imperative that you have the programming all done on paper, line by line, before trying to load it into the computer. Sitting at the keyboard wasting electricity is no time to be figuring out the lines of a program. But adventure programs are different. Once you have one finished program in hand, you can use that as your outline for the next one. Only the names are changed to protect the innocent. With a good, complete design in mind, you can actually do most of your programming at the keyboard. Not that it will come out perfect the first time—you can't expect that kind of success with even a one-line program!—but with the techniques I'll describe later you should get it running like a charm by the third run-through, not a bad debugging experience. You might set aside about three months of evenings for the entire process.

The first step in programming, after setting the movable objects (as we did in Chapter 1), is laying out the rooms. You could conceivably start out somewhere else in the program, but I wouldn't advise it. When a player is going through an adventure he spends the majority of his time simply poking through the various rooms, looking at things and examining them, going back and forth trying to get the feel of the game universe. The computer conversation segment of the module must be as comprehensive as possible in predicting exactly what the player will ask about. Doing the room layouts first allows you to have in hand an exact copy of what the player will see in his travels, allowing you to handle the conversation segment accordingly. The better you handle this at the start the less time you'll have to spend debugging it later.

For convenience's sake I start my room segment at line 50000. If the speed of the program were at issue this might not be the best place to put this. The room segment is a frequent stop in the program, and the more frequently a routine is used in a program the closer to the beginning of the program you should locate it. Whenever you put a branch into a program, the computer starts at the first line of the program to find it (e.g., if you're at 20000 and you have a command GOTO 50000, the computer begins looking for 50000 way back at statement 1). If a program does a lot of variable sorting at high program lines, you could be seriously hindering your computing capabilities. As it is, there is a pause for a fraction of a second in the "Space Derelict!" program whenever there's a branch to the room segment—there's an eye blink's delay as the old room information is cleared and the new information is displayed. If you were to put enough business between line 50000 and the branch directions of line 51240, you could possibly create a disturbing delay in the play. This concept of flow and speed is important to keep in mind as your programming becomes more sophisticated, but for our purposes here I'm more interested in clarity than celerity. We'll discuss processing speed in more detail in Part Two.

Before going further, let's get the actual program on the boards. Remember, if you want to save "Space Derelict!," you'll have to type in these lines. Here is the room layout routine.

```
50000 REM  ROOMS
50010 GOSUB 13000
50020 POKE 34,0
    : POKE 35,13
    : HOME
50030 T = 0
50040 ON R GOTO 50050,50100,50150,50200,502
        40,50290,50330,50390,50440,50490,50
        540,50590,50640,50640,50710,50760,5
        0800,50840,50590,50880,50930,50990,
        51060,51170
50050 PRINT "THE ONLY ENTRANCE TO THE SHIP.
        "
    : PRINT
50060 PRINT "AN OPEN AIRLOCK LEADS BEYOND M
        Y LINE OF SIGHT."
    : PRINT
50070 PRINT R$
    : PRINT
50080 PRINT "<----------------------------
        --------->"
50090 GOTO 51210
50100 PRINT "AIRLOCK."
    : PRINT
50110 PRINT "DOOR APPARENTLY LEADING INTO S
        HIP. PANELOF THREE BUTTONS (BLUE, B
        LACK AND GREEN)NEXT TO DOOR."
    : PRINT
50120 PRINT R$
    : PRINT
50130 PRINT "<----------------------------
        --------->"
50140 GOTO 51210
50150 PRINT "APPARENTLY EMPTY ROOM."
    : PRINT
50160 PRINT "THREE DOORWAYS."
50170 PRINT R$
    : PRINT
50180 PRINT "EXITS:EAST, WEST, SOUTH."
    : PRINT "<----------------------------
        --------->"
50190 GOTO 51210
50200 PRINT "LARGE ROOM. SOME SORT OF PENTA
        NGULAR    ELECTRONIC DEVICE IN CENT
        ER OF ROOM. A  LARGE HOLOGRAM TOWAR
        D EASTERN WALL."
    : PRINT
50210 PRINT R$
    : PRINT
50220 PRINT "EXITS: WEST, SOUTH."
    : PRINT "<----------------------------
        --------->"
50230 GOTO 51210
```

```
50240 PRINT "LARGE ROOM WITH PEDESTAL IN CE
        NTER."
    : PRINT
50250 PRINT "SOME SORT OF DESTRUCTIVE DEVIC
        E SEEMS TOBE PLANTED IN THE CEILING
        ."
    : PRINT
50260 PRINT R$
    : PRINT
50270 PRINT "EXIT: NORTH."
    : PRINT "<-----------------------------
        --------->
50280 GOTO 51210
50290 PRINT "LARGE APPARENTLY EMPTY ROOM."
    : PRINT
50300 PRINT R$
    : PRINT
50310 PRINT "DOORS: NORTH, SOUTH, EAST."
    : PRINT "<-----------------------------
        --------->"
50320 GOTO 51210
50330 PRINT "LARGE ROOM."
    : PRINT
50340 PRINT "PEDESTAL IN CENTER OF ROOM. PA
        NEL OF"
50350 PRINT "THREE BUTTONS (BLUE, BLACK AND
         GREEN)   NEXT TO DOOR."
    : PRINT
50360 PRINT R$
    : PRINT
50370 PRINT "DOOR: NORTH."
    : PRINT "<-----------------------------
        --------->"
50380 GOTO 51210
50390 PRINT "LONG PASSAGEWAY HEADING NORTH/
        SOUTH."
    : PRINT
50400 PRINT "FOOTPRINTS (?) IN DUST LEAD NO
        RTH."
    : PRINT
50410 PRINT R$
    : PRINT
50420 PRINT "EXITS:NORTH, SOUTH."
    : PRINT "<-----------------------------
        --------->"
50430 GOTO 51210
50440 PRINT "CROSSROADS IN PASSAGEWAY."
    : PRINT
50450 PRINT "FOOTPRINTS LEAD WEST."
    : PRINT
50460 PRINT R$
    : PRINT
50470 PRINT "EXITS: WEST, EAST, SOUTH."
    : PRINT "<-----------------------------
        --------->"
```

```
50480 GOTO 51210
50490 PRINT "END OF PASSAGEWAY."
    : PRINT
50500 PRINT "FOOTPRINTS LEAD TO WESTERN DOO
        R."
    : PRINT
50510 PRINT R$
    : PRINT
50520 PRINT "DOORS: NORTH, SOUTH, WEST.+ EX
        IT EAST."
    : PRINT "<----------------------------
        --------->"
50530 GOTO 51210
50540 PRINT "PASSAGEWAY INTO ROOM SIMILAR T
        O THE LASTONE."
50550 IF CO(11) = 105 THEN PRINT "A TWISTED
        , DESTROYED METAL OBJECT, SOME SORT
         OF ALIEN ROBOT PERHAPS, IS VISIBLE
         JUST INSIDE THE ROOM."
50560 PRINT R$
    : PRINT
50570 PRINT "EXITS: NORTH, SOUTH."
    : PRINT "<----------------------------
        --------->"
50580 GOTO 51210
50590 PRINT "CREWS QUARTERS (?)."
50600 PRINT "NUMEROUS PENTANGULAR BEDS (?)
        OR         SIMILARLY CONSTRUCTED ITE
        MS. STORAGE     SPACE, CLEARED OF AL
        L OBJECTS."
    : PRINT
50610 PRINT R$
    : PRINT
50620 PRINT "DOOR: NORTH."
    : PRINT "<----------------------------
        --------->"
50630 GOTO 51210
50640 PRINT "APPARENT ENGINE ROOM."
    : PRINT
50650 PRINT "ACCELERATOR TUBES SIMILAR TO T
        HOSE OF    KRANTZ/WARREN HYPERSPACE
        DRIVE."
    : PRINT
50660 PRINT R$
    : PRINT
50670 IF R = 13 THEN PRINT "DOOR: EAST."
50680 IF R = 14 THEN PRINT "DOOR: WEST."
50690 PRINT "<----------------------------
        --------->"
50700 GOTO 51210
50710 PRINT "APPARENT MAIN CONTROL ROOM."
50720 PRINT "NAVIGATIONAL PANEL WITH BUTTON
        S. PEN-    TANGULAR CHAIRS WITH COMP
        UTER CONSOLES. PEDESTAL TOWARD SOUT
        HERN END."
```

```
50730 PRINT R$
50740 PRINT "DOORS:EAST(N), EAST (S), WEST(
      N),       WEST(S)."
    : PRINT "<-----------------------------
      --------->"
50750 GOTO 51210
50760 PRINT "PASSAGEWAY ENDING IN DOORWAY E
      AST."
    : PRINT
50770 PRINT R$
    : PRINT
50780 PRINT "EXITS: WEST, EAST."
    : PRINT "<-----------------------------
      --------->"
50790 GOTO 51210
50800 PRINT "PASSAGEWAY."
    : PRINT
50810 PRINT R$
    : PRINT
50820 PRINT "EXITS: EAST, WEST."
    : PRINT "<-----------------------------
      --------->"
50830 GOTO 51210
50840 PRINT "PASSAGEWAY EAST/WEST."
50850 PRINT R$
    : PRINT
50860 PRINT "EXITS: EAST, WEST, NORTH (DOOR
      ),         SOUTH (DOOR)."
    : PRINT "<-----------------------------
      --------->"
50870 GOTO 51210
50880 PRINT "PASSAGEWAY ENDS AT EASTERN WAL
      L."
50890 PRINT R$
    : PRINT
50900 PRINT "EXIT: WEST."
50910 PRINT "<-----------------------------
      --------->"
50920 GOTO 51210
50930 PRINT "APPARENT KITCHEN."
50940 PRINT
50950 PRINT "VARIOUS STEEL TABLES, SCULLERY
      , CON-    VECTION OVENS, LARGE CLOS
      ET."
    : PRINT
50960 PRINT R$
    : PRINT
50970 PRINT "DOORS: NORTH, SOUTH."
50980 PRINT "<-----------------------------
      --------->"
    : GOTO 51210
50990 PRINT "APPARENT EXERCISE AREA."
    : PRINT
51000 PRINT "MARKED-OFF BALL(?) COURT."
51010 IF CO(12) = 0 THEN PRINT "LONG CLIMBI
      NG (?) ROPES HANGING DOWN."
```

```
51020 PRINT "LARGE SWIMMING (?) POOL, FROZE
      N SOLID."
51030 PRINT R$
    : PRINT
51040 PRINT "DOORS: NORTH, SOUTH."
    : PRINT "<-----------------------------
      --------->"
51050 GOTO 51210
51060 IF L = 0 OR CO(8) < > 1 THEN
      INVERSE
    : FLASH
    : PRINT "IT'S TOO DARK TO SEE IN HERE!"

51070 IF CO(8) < > 1 THEN NORMAL
    : GOTO 51210
51080 IF L = 0 THEN FOR PAUSE = 1 TO 2000
    : NEXT
51090 IF L = 0 THEN PRINT "I'M TURNING ON M
      Y FLASHLIGHT. PLEASE    STAND BY."
    : FOR PAUSE = 1 TO 2500
    : NEXT
51100 L = 1
51110 HOME
    : NORMAL
51120 PRINT "ENORMOUS ROOM STRETCHING ALL T
      HE WAY TO WHAT MUST BE THE FRONT OF
       THE SHIP."
51130 PRINT
51140 PRINT R$
    : PRINT
51150 PRINT "DOORS: NORTH, SOUTH (TO THE WE
      ST),       SOUTH (TO THE EAST)."
    : PRINT "<-----------------------------
      --------->"
51160 GOTO 51210
51170 PRINT "TRIANGULAR CRYSTALLINE ROOM."
    : PRINT
51180 PRINT "A CONTROL PANEL OF SOME SORT A
      T THE      FRONT WITH A SLOT IN IT."

    : PRINT
51190 PRINT R$
    : PRINT
51200 PRINT "EXIT: SOUTH."
    : PRINT "<-----------------------------
      --------->"
51210 POKE 35,24
    : POKE 34,13
51230 HTAB 1
    : VTAB 24
51240 GOTO 3000
```

Now you're ready to recall our discussion of the room variable, R. R had a value from 1 to 24, and the value of R told us which of the 24 rooms we were in. Now you'll see exactly how that works.

When the puppet robot enters a given room, the monitor displays a variety

of information in a fairly standard fixed format. The top of the screen describes the room and lists the objects in it and the exits out of it. Beneath this information is space for the conversation with the computer. This information at the top of the screen does not scroll; i.e., every time a new line appears at the bottom, the top remains unchanged—until a new room is entered. The only time the top changes is when movable objects are either left behind or picked up: The top of the screen will reflect these transactions. The video screen, simply enough, is divided in two: constant data at the top, changeable data at the bottom, which scrolls up and disappears in the middle. That collection of data at the top is the concern of the 50000 room-layout routine.

Line 50000 does nothing but tell us where we are in the program. The adventure module doesn't have to be loaded with a lot of REM statements, and it isn't. REMs are another way of slowing down a program, but they are useful in keeping track of what you're doing. They are especially useful when you set your variables and at the beginning of subroutines. The more complex your program, the more likely you'll need a lot of notes to yourself detailing what you're doing, and hence the more REMs you'll want to use. If other people are going to be looking at your program, REMs become that much more important and should be used that much more liberally. Since I've written a book to explain the "Space Derelict!" program, I've haven't REMed it that much.

Line 50010 branches to the room-objects subroutine at 13000, which we'll get to shortly. It figures out what's in each room (CO(X) either equals 100+R or it doesn't) and reports its findings in a new variable, the string R$. Since 50010 is a GOSUB, when the routine at 13000 is completed, processing continues at the next line after 50010.

Line 50020 does the business of setting the text window, the dividing of the video screen in two as described above. We'll have to go into this at length to explain it clearly.

When you first turn on your computer, the output sent to the video screen is formatted in a pattern 40 characters across and 24 lines down (unless you have a special 80-column piece of hardware you paid extra for; 80-column versus 40 makes no difference in this particular discussion. Pick your poison.). This 40 (or 80)×24 box is known as the text window, and it is not inflexible. We can adjust it to our own needs, left and right and up and down.

In our adventure we want to adjust the window so that the information at the top, describing a room, stays still while the information below, the conversation with the computer, scrolls up and away. Line 50020 is half of how we do that. POKE X,Y tells the computer to put the value Y into machine address X. The 34 in POKE 34,0 is the address that governs setting the window from the top down; the second number tells it precisely how many lines down to go. Since that second number here is 0 (which is the normal, or default, setting), POKE 34,0 in effect clears the text window to the default measurement of 24 lines to the bottom; now the screen will scroll in a normal fashion. But let's assume that we are in room 20, we've been conversing with the computer, and now we are entering room 18. We want to change the display at the top of the screen but leave the recent conversation unchanged. We have to change the top of the screen without changing the bottom. That's where POKE 35,13 comes in. The machine address 35 gov-

erns setting the window from the bottom up; the second number (which here is 13) tells the computer how many lines *down from the top* we're going to set that bottom of the window (sorry about that, but I didn't invent the machine). So now, with the setting of POKE 35,13, we have an imaginary line running roughly across the middle of the screen. The data *above* the line will scroll normally while the data *below* the line will remain etched in electronic stone.

What happens when the HOME command is executed on this same line 50020? Simple: the top half of the screen is cleared since it's still scrolling normally, but the bottom half remains unchanged. Hey presto! The main portion of the 50000 routine is dedicated to filling up that top half of the screen, but let's skip down to line 51210 before getting into that. After we've written in our room description, POKE 35,24 removes the imaginary line we set with POKE 35,13: 24 is the bottom line of the screen, so POKE 35,24 means no more unsightly ring around the middle, and the window is now at default, 40×24 again. POKE 34,13, on the other hand, resets the window down from the *top*. So now all the information on the top will remain etched in electronic granite while the data on the bottom will scroll normally. Hey presto again, we've very cleverly sawed the volunteer from the audience in two.

Some adventures do not use this style of separating room descriptions from the conversation (the very sophisticated Infocom adventures such as "Zork" are among those nonseparated games). This would actually make for simpler programming in that you would delete all this window business entirely, allowing the room descriptions to scroll with the conversation. But you would also need to rewrite your room layouts in a more conversational tone than we've used here. It's mostly a matter of personal preference; if you like it this way, fine. If you like it the other way, simply eliminate the window changes. We'll talk more about this later. In either case, it's good to know how the window works.

Ignore line 50030 for the time being. Program it, but forget about it. Its day will come. And welcome to line 50040.

ON R GOTO . . . Now, that's pretty ingenious, and it's one of the nicest Basic commands we have. As I've repeated *ad nauseam*, the variable R tells us what room we're in. This line of the program tells the computer how to do something about that. ON statements evaluate the variable named and then branch to the appropriate program number as listed. So if R=1, then the program will branch to 50050, the first number listed after the ON; if R=2, then the program will branch to 50100, the second number, and so on, up to a value of R=24 and a branch to 51170. Very nifty, that. You'll notice that there are no fail-safes here for values of R less than 1 or greater than 24. The program is designed so that other values of R are impossible, so we don't have to worry about that. Fail-safes would be necessary if that were not the case, about which more later.

All these ON . . . GOTO branches from line 50040 are the physical descriptions of the rooms, in order, from room 1 to room 24. We'll examine three of them and leave the rest to your imagination, since they are all basically the same.

Line 50050 gives the name of the room, followed by a blank line (the

dataless PRINT statement). Line 50060 further details the room, giving pertinent descriptions, again followed by a blank. Line 50070 prints R$; R$ was obtained in the subroutine at 13000 to which we branched from line 50010. R$, to recapitulate, is a list of the movable objects lying around in the room. Another blank separates us from 50080, a real line above the imaginary line of the text window. Line 50090 takes us to 51210, the end of the 50000 room routine.

We'll skip down to line 50200, which describes room 4. The description is slightly different from the one for room 1 in that it is a little wordier and also contains exit information. Line 50200 gives the overall description of the room and describes some of its particulars. Line 50210 prints R$, the movable object contents of *this* room (you'll see how R$ can be so flexible as to work in any room when we discuss the R$ subroutine). Line 50220 has the new business of the exits: It lets the player know in which directions he may go (this was unnecessary in room 1, since the only place to go was the airlock). Line 50230 branches to 51210, again, the end of the routine.

The only real straying from this room-programming setup occurs in lines 51060–51110. This is the beginning of room 23, and here I've turned out the lights to give you some idea how that can work. The variable L in 51060 refers to the robot's flashlight, which is either off or on, for respective values of either 0 or 1. If it's off, then we get the flashing message that it's too dark to see. This line also checks to make sure the flashlight is actually in the robot's hand (the light could easily be on but be in another room!): If it's not in hand, then we still want the "too dark" message. Line 51070 takes us out of the rest of the description of room 23 if in fact the robot is not carrying the flashlight—it's too dark to see, so we're not going to see anything. Lines 51080–51090 allow our puppet robot to show a little initiative on his own. It's dark, so he turns on his flashlight himself. Now that the light is on, L = 1, and we segue into a normal room description.

All the rest of the rooms conform to the normal structure, with occasional bows to the exigencies of space. Having set up a constant 13 lines to work in (POKE 34,13—remember?), there's just so much you can fit in. Complicating matters quite a bit is the R$ command. If no movable objects are lying around and R$ therefore equals "" (i.e., zilch), then R$ takes up just one (empty) line. But if there are half a dozen objects lying around, R$ will take up three lines—hence the somewhat rubbery structure of the room designs. And, as I mentioned earlier, some judicious editing—namely, putting one arrow line (such as 50080) at the end rather than 24 separate ones—could be applied here as well. But I thought it might be easier for you to see it all laid out in black and white in front of you.

We've already discussed 51210, the resetting of the window. Line 51230 deals with the location of the cursor. After all the business of changing rooms is done, we want to go back to conversation with the computer, and we want that conversation to begin at the bottom of the screen. HTAB 1 sets the cursor at the left of the horizontal, and VTAB 24 sets the cursor at the bottom of the vertical—the two together putting the cursor at the very bottom left-hand corner of the screen. And 51240 brings us back to the routine at 3000, which is the conversation portion of the program.

And we've now got all our rooms laid out.

## SWEATY PALMS AT THE KEYBOARD

As I said earlier, I don't find it necessary to write out all the lines of an entire adventure before programming it now that I have the basic structural module to work from. In fact, after I design an adventure with maps and lists of objects and rooms and so forth, I don't write anything else on paper. My procedure is exactly the one I am laying out here. First I make up a list of the movable objects and enter that list with locations: I give names to all the CO$(X) variables, and I give numbers to all the CO(X) variables. Then I enter these program lines somewhere around 2000 (we'll get into why later). Then I begin with the rooms. I start at 50000 with the unnecessary REM and proceed from there. Line 50010 is always there, as is line 50020 (50030 is special to "Space Derelict!" and didn't find its way to the listing until almost the end). Then I enter the first part of 50040 thusly:

50040 ON R GOTO 50050,

period. I know that 50050 is the next line where I'll begin describing room 1, and that's all I know. Then I lay out room 1 exactly as you see it, but line 50090 would look like this:

50090 REM NEED EXIT ADDRESS

At this point I don't know what the last line of the module will be, and I won't know until I get there. All the exit statements from the other rooms were referenced in this same way, waiting for the actual exit to come into being for insertion later.

When I finished the first room I knew that 50100 would be the first line of room 2, so I went back and edited line 50040 thusly:

50040 ON R GOTO 50050,50100,

To do this I used the J-K-I-M keys as described in the Applesoft manual. I can't explain this editing function any better than the manufacturer did, and it is another one of the really nice features of the Apple. If you don't understand how to use these keys, drop everything and find out. It's one of the most important time-saving devices known to modern man.

I go along with the whole room-layout section in this manner, slowly adding each room and appending line 50040 until I finally have all the rooms described in complete detail. Then—and this is crucial—I run off a printout of what I've done. It would be practically impossible to design the conversation segment of the program without the room layouts in hand. Most of what the player is going to ask you about is the stuff you've described in your rooms, and your program has to be prepared to answer all those questions. You can't describe a room as having a bearskin rug in it and then not have an answer in your program to the requests LOOK RUG or GO RUG or TAKE RUG. You could wait until you're in the debugging phase to handle this sort of thing, but why make life harder on yourself? Do it now, in the words of the prophet.

# 3

# SETTING THE SUBROUTINES

## THE INTRODUCTION

There are two things you have to do at the outset when the player sits down at the computer to play your game. First you have to tell the player what the game is all about, and then you have to tell the computer what the game is all about. It's a lot easier to explain it to the player than to the computer.

The introduction to the game is the first thing that the player will see when he runs the program, and it should fulfill two purposes. First, it should explain briefly what adventuring is all about, in case you're dealing with a completely novice player. Second, it should set the scene for your adventure, just as an opening chapter sets the scene in a novel. You could just drop your player into the middle of your universe without any warning, but it's a lot nicer to lead him in slowly and deliberately. And there is always the possibility of hiding a few clues in your introduction. This is the only direct communication you will have with the player before the program takes over, and you might just be able to put it to good use above and beyond the call of introducing.

In explaining the play of the game you give the player some idea of how to conduct his end of the conversation with the computer. You describe how the computer understands various one- and two-word commands and perhaps give a few of the more common examples (TAKE, GO, SAVE GAME, etc.). If you're incorporating any special features into the game (using special abbreviations for frequent commands such as INV for INVENTORY or that sort of thing), this is the place to explain them. But don't get too carried away telling the player what to do. Half the fun of adventuring is figuring out what was going on in the mind of the game designer, so you don't want to make your mind an open book.

The other half of the introduction is setting the scene. This is your chance to get really dramatic in a sense not possible in the actual game. You want to pull your player in, tease him, get him interested, and most of all, make the game that follows *real*. Unfortunately, this is a part of the game a lot of designers overlook.

Seeing how the scene was set in "Space Derelict!" will give you some idea of how you might do it yourself.

***"Space Derelict!"***

```
10    DIM A$(23),CO$(16),CO(16)
20    DIM D(17)
30    TEXT
40    HOME
50    GOSUB 15000
    : REM  INTRODUCTION
60    HOME
70    GOSUB 2000
    : REM  SET DIMS
80    INPUT "DO YOU WISH TO PLAY A SAVED GA
        ME?";C$
90    IF C$ = "Y" OR C$ = "YES" THEN
        GOSUB 31000
    : GOTO 50000
100   IF C$ = "N" OR C$ = "NO" THEN 120
110   GOTO 80
120   HOME
130   INVERSE
140   G$ = ""
    : FOR N = 1 TO 10
    : PRINT G$
    : NEXT
150   SPEED= 100
160   PRINT "SERIES R ROBOT EXPLORER NUMBER
         182-X3   CALLING COMPUTER CONTROL.
         I HAVE AN     UNIDENTIFIED FLYING
       OBJECT IN RADIOMETRYRANGE. I AM GOI
       NG IN. PLEASE STAND BY."
170   FOR PAUSE = 1 TO 1000
    : NEXT
180   PRINT
190   PRINT "UNIDENTIFIED FLYING OBJECT IS
        A FIVE-   SIDED INTERGALACTIC SPACE
        CRAFT. UFO DOESNOT RESPOND TO MY ME
        SSAGES. I AM LEAVINGEXPLORER CRAFT
        TO EXAMINE MORE CLOSELY."
200   PRINT "I WILL BE TURNING COMMAND OVER
         TO COM-  PUTER CONTROL. I AM TURNI
       NG MY TRANS-   MITTER ON HIGH TO SP
       EED UP TRANSMISSION TIME. PLEASE ST
       AND BY."
210   FOR PAUSE = 1 TO 2000
    : NEXT
220   NORMAL
230   HOME
```

```
240   SPEED= 255
250   GOTO 50000
15000 PRINT "              ";
15010 INVERSE
15020 PRINT "SPACE DERELICT!"
15030 NORMAL
15040 PRINT
15050 PRINT "YOU ARE OPERATING A COMPUTER S
        TATION    LOCATED ON THE SOL SIDE O
        F THE PLANET   PLUTO. YOU ARE THE M
        ANAGER OF A PLATOON OF SERIES R EXP
        LORER ROBOTS."
    : PRINT
15060 PRINT "SERIES R ROBOTS ARE PROGRAMMED
         TO TRAVELTHE VAST EXPANSES OF EXTR
        A-GALACTIC      SPACE, REPORTING ON
        PHENOMENA THEY       ENCOUNTER.
15070 PRINT
    : PRINT "UNDER NORMAL CIRCUMSTANCES SER
        IES R      ROBOTS ARE SELF-OPERATING
        . HOWEVER, IN  EMERGENCIES OR UNUSU
        AL SITUATIONS THE   CONTROL IS TAKE
        N OVER AT THE COMPUTER   STATION."
15080 PRINT
    : PRINT "SERIES R ROBOTS RESPOND TO A L
        ARGE       NUMBER OF SIMPLE ONE AND
        TWO WORD        COMMANDS. THESE INCL
        UDE GO, TAKE, LOOK, SAVE GAME, ETC.
        "
15090 GET C$
15100 PRINT
    : PRINT "ON DEPRESSING THE RETURN FUNCT
        ION OF    YOUR CONSOLE YOU WILL TAK
        E OVER COMMAND OF THE COMPUTER CONT
        ROL STATION. CARRY  ON, LIEUTENANT.
        "
15110 GET C$
15120 RETURN
```

Lines 10–50 are the first lines in the program. With the exception of A$(X), which we'll get to in a minute, you've already met up with the variables in lines 10 and 20. Dimensioning, like Christmas shopping, is best done early—this is a common and recommended programming practice. By the way, line 10 shows how you can easily dimension more than one variable on one line by the use of commas after each index.

Line 50 branches to the subroutine at 15000, which we'll look at now. This subroutine is simply a collection of PRINT statements that primarily explain how to play an adventure but in a small way begin to set the scene as well. Lines 15000–15040 simply get the words "Space Derelict!" at the top of the screen in inverse printing. The formatting style here is fairly primitive—I just counted out the spaces as I typed—but it does work, and in most cases, if something works, that's all you need to know. Notice the semicolon at the end of 15000. Under normal conditions, at the end of a line of PRINT programming your computer executes an imaginary carriage re-

turn (as in Return button), just like a typewriter carriage rolling up at the end of each typed line. The semicolon tells the computer *not* to issue a Return. As a result, the next PRINT statement will immediately follow the last one, one cursor position away from the last character PRINTed.

Lines 15050–15080 are self-explanatory. The occasionally odd spacings result from making sure that the 40-column text doesn't break in the middle of a word. It is very easy to keep track of this when you are entering a program. When you actually type in a line such as:

```
100 PRINT "MANY BRAVE MEN ARE ASLEEP IN
THE DEEP, AND NONE OF THEIR CHICKENS
COULD CARE A BIG PEEP"
```

you are seeing the line printed 40 characters across even as you type it—unless you have 80-column hardware, where you see it printed 80 characters across. So all you have to do is keep an eye on the first pair of quotation marks, being careful not to cross through the column that contains them as you do your typing. It will all come up as clean as a whistle as a result.

You can see that these lines in the program set up the concept of the player as Mission Controller of a platoon of robots. And in 15080, there's a brief explanation of how to command those robots, which act as instructions to the player. The material up to this line fills the whole screen, so the GET statement in 15090 simply scrolls up on any keyboard input to print 15100—the C$ part of it is meaningless and simply fulfills the syntactical obligation of the GET command (see the Glossary if you don't understand GET). The GETs in this routine allow the player to read at his own speed. The second GET takes us to the end of the subroutine and the RETURN in 15120, which brings us back to line 50.

The subroutine branch from line 70, which sets the variables, will be covered in the next section, on housekeeping. Line 80 gives the player the opportunity to retrieve an old game to take up where he left off. Once again you can see the C$ variable; this is the one I usually employ to cover miscellaneous input from the user no matter what the program (AN$ as in ANswer is a common variation of this). It's a good idea to keep some consistency in what you do from one program to another to make your own work easier. If a certain type of variable is always represented by a particular symbol, you'll always know what you're talking about no matter what program you're working on. With so many *really* variable items in any given program, it's nice to have a few variables that remain constant.

If a player does wish to play a saved game, line 90 will branch to the subroutine at 31000, which will be discussed later in this chapter. This routine fills in the saved values for all the game variables, thereby allowing the player to take up where he left off. After the 31000 GOSUB is completed, the program branches off to 50000, the room segment (and from there on to the conversation with the computer). Line 100 continues the player through the introduction if he does not wish to play a saved game. Line 110 sends the player back to line 80 if an answer other than "YES", "Y", "NO", or "N" is given. Which leads us into the subject of foolproofing your programs.

No matter how clearly you explain something in a program, no matter how carefully you prompt the user for a particular brand of information, sooner or later some answer is going to be put in other than the information

you were asking for. It is absolutely essential any time you request input from the user that you make sure the program can handle it, no matter what the input is.

Since it is impossible to predict every idiosyncracy a user can come up with, you have to figure out all the correct inputs and allow for them, and by process of elimination disallow any others. In an adventure, where we're dealing entirely in string input, we have to have adequate responses for all the right answers, and adequate responses for a number of wrong answers too, and the programming sometimes can become a bit complicated as a result. In strategy games, where a move is either allowable or it isn't, things are a lot simpler.

In the case at hand in lines 80–110, we have a simple situation where we are looking for either a YES or a NO answer. But even here we get a little more elaborate than that, allowing both for the complete words YES and NO as well as the shortened Y and N (it is this sort of touch that makes for a more responsive, "intelligent" game, where the player can concentrate on solving your complications rather than solving your programming lapses). If none of these values is input, then the program branches back to line 80 until the player gets it right.

The rest of the introductory routine continues orienting the player to the game universe, and here I get a bit more dramatic, using all sorts of simple tricks to get things heated up. First off, we go into INVERSE text in line 130. Line 140 is very mysterious until you learn that there's a CONTROL G hidden in those quotation marks—CTRL-G rings the little imaginary bell inside the computer, and this programming line rings it 10 times (for an explanation of FOR . . . NEXT commands, see the Glossary). And as if that isn't enough, line 150 resets the print speed of the monitor to 100, about as fast as a moderately capable typist (for more on SPEED, again see the Glossary). So the rest of the PRINT statements are sent to the screen almost as if they were coming across on a teletype machine. The reason? This is our puppet robot talking, good old Series R Explorer Number 182-X3. He's way out in outer space sending back a message directly to your computer console. Why not put a little spice into it? The message is self-explanatory, and lines 220–240 reset the video monitor back to normal. Then line 250 branches off to the room-layout routine at 50000. Since the screen has been cleared in line 230, when the description of room 1 is displayed the portion at the top of the screen will be filled with the description of that room, and the bottom of the screen will be empty, which is exactly what we want (the value of R will have been set in the housekeeping routine that branched from line 70).

The main point to remember about your introductory routine is that this is a chance to have a little fun, and you might as well make the most of it.

## HOUSEKEEPING

Almost every program needs someplace off the beaten path where variables can be set without colliding into any of the actual processing. This could be done just about anywhere in a program; as a rule it is best to put this material at the end, since it is a one-access-only routine, and assigning it the high

listing numbers won't affect processing speed and also will free those lower listing numbers for your more frequent accesses (remember, when you branch from a GOTO or GOSUB, the computer starts at the first listing in the program every time). However, since speed isn't much of a factor in an adventure game, I have put the housekeeping at the lines beginning at 2000.

There aren't many variables in an adventure, so the housekeeping doesn't take up very much space—in the case of "Space Derelict!" only about 45 lines or so. For the most part these variables are radically different from game to game, depending, for instance, on what your movable objects (CO$(X)s) actually are. So let's go right into the specifics, which will set you up for your own parallels.

***"Space Derelict!"***

```
2000  REM  SET VARIABLES, HOUSEKEEPING
2010  R = 1
2020  REM  A$=ANSWERS FROM COMPUTER
2030  A$(1) = "INSTRUCTION NOT ACCEPTABLE.
        PLEASE       CLARIFY."
2040  A$(2) = "I'M SORRY, BUT I CANNOT RESP
        OND TO THE  COMMAND "
2050  A$(3) = "OKAY."
2060  A$(4) = "I CANNOT GO IN THAT DIRECTIO
        N."
2070  A$(5) = "THERE IS A LOCKED DOOR IN TH
        AT DIRECTION"
2080  A$(6) = "I DON'T HAVE THE FACILITIES
        TO DO THAT  AT THIS TIME."
2090  A$(7) = "THAT ITEM IS NOT HERE AT THI
        S TIME."
2100  A$(8) = "TELL ME HOW."
2110  A$(9) = "I'M NOT QUITE SURE WHAT YOU
        MEAN. COULD YOU BE MORE SPECIFIC?"
2120  A$(10) = "IT SEEMS TO BE SOME SORT OF
         MACHINE PARTOR SIMILAR ITEM. IT FI
        TS EASILY IN MY   HAND."
2130  A$(11) = "IT IS THE SIZE OF A MATCHBO
        OK. IT LOOKS VERY MUCH LIKE A CHILD
        REN'S TOY. IF ONE WERE TO 'CLICK' I
        T, IT WOULD MOST LIKELYMAKE A LITTL
        E CLICKING SOUND."
2140  A$(12) = "IT IS ABOUT FIVE INCHES ACR
        OSS. THE      LIGHT PLAYS AROUND IN
        IT AS IF IT HAS A LIFE OF ITS OWN."
2150  A$(13) = "IT IS A NON-EARTH ORIGIN PH
        ASER GUN. IT IS NOT FIVE SIDED. IT
        APPEARS TO BE NON-FUNCTIONAL."
2160  A$(14) = "IT IS SILVER. IT OPENS EASI
        LY. IT IS    ABOUT SEVEN INCHES ACR
        OSS."
2170  A$(15) = " IT IS THE CRYSTALLINE STRU
        CTURE OF A    SUBSTANCE NOT KNOWN O
        N EARTH. IT IS      ABOUT 7 INCHES E
        ND TO END."
```

```
2180  A$(16) = "IT'S THE STANDARD ISSUE SER
        IES R FLASH."
2190  A$(17) = "IT IS ABOUT A FOOT LONG WIT
        H A STEEL    CUTTING EDGE OF ABOUT
        8 INCHES."
2200  A$(18) = "IT APPEARS TO BE MADE OF AN
         ALUMINUM-   TYPE ALLOY."
2210  A$(19) = " IT IS ROUGHLY HUMANOID IN
        SHAPE. IT     APPEARS TO HAVE BEEN
        BLASTED BY SOME    SORT OF HEAT WEA
        PON."
2220  A$(20) = "IT IS SOMEWHAT SPIDERLIKE A
        ND DEFINITELYLETHAL."
2230  A$(21) = "NOTHING HAPPENS."
2240  A$(22) = "I DON'T SEE ANYTHING REMARK
        ABLE."
2250  A$(23) = "IT'S THE STANDARD COLT LASE
        R ISSUED TO  ALL SERIES R ROBOTS."
2430  L = 0
    : REM  L IS FLASHLIGHT, L=1 MEANS ON
2440  W = 2
    : REM  LIMIT AMT OF INVENTORY TO 5
2450  REM  FOR N=1 TO 17, ALL D(N)S = 0
2460  REM  AL IS AIRLOCK,0 CLOSED, 1 OPEN
2470  RETURN
```

Line 2010 sets R at 1 so that when we hit the room-layout routine the first time around we'll have someplace to go vis-à-vis the ON statement in 50040. Line 2020 explains what A$ is all about (you'll recall that we DIM'd A$ for 23 values back in line 10)—namely, the computer's responses to the players' input. Almost the entire content of an adventure game is conversation with the computer, and in the routine at 3000, which we'll discuss in the next chapter, the program attempts to predict all the possible vocabulary input from the player. The conversation routine also directs the computer to print responses to the player's input. If we were so inclined, in the conversation routine we could print out the entire response to each and every player input in each and every instance, but it is a lot easier if we have a virtually complete set of answers already made, so that all we have to do is type in a variable response. It is a lot easier to type:
PRINT A$(6)
100 times than it is to type:
PRINT "I DON'T HAVE THE FACILITIES TO DO THAT AT THIS TIME"
100 times. We'll postpone a complete discussion of A$ until we get to Chapter 4, "Conversation with the computer," but you might as well enter these listings now that they're right in front of you.

Lines 2260–2410 were covered in Chapter 1 and are not repeated here. These were the assigning of the strings to the CO$(X) variables and the placing of the CO(X)s.

Line 2420 has been deleted in the debugging process, so don't get upset that you're missing a listing—I've tried to mainstream 10-increment lines throughout but sometimes it just wasn't possible. Line 2430 is self-explanatory. If the flashlight is on, L has a value of 1; if it's off, L has a value of 0.

The need for a flashlight comes up in room 23, where it is too dark to see without it. The need for this line of text is a little less obvious. If no value is originally assigned to a variable, the computer will automatically assign to that variable a default value of 0. So I don't really need to set L as 0 here, since the computer will do it for me the first time my unvalued L pops up. The point is that you might find it a good idea to set *all* your variables during housekeeping time, even if their values are null. Line 2430 could have been done without the L=0 part, but it still would have been imperative to leave in the REM. This way you know that when you look at your housekeeping routine there will be a mention of each and every variable regardless of its value. This is extremely important when it comes to putting together the game-saving routine (and also if you want to sell a program—software publishers like lists of variables in the documentation, and if you have variables in your program that aren't in your documentation . . . good-bye, Charlie).

W in line 2440 stands for weight. This is how we limit the inventory. If we wanted we could have no such limitation and allow our puppet robot to carry everything he can get his hands on, but this is both unrealistic and a little too easy. In "Space Derelict!" there is a weight limitation of five items (the actual limiting of the weight takes place in the conversation-with-the-computer segment). If you wanted, you could get much more complex than this. You could assign a literal weight to each movable object and then limit your puppet to a certain poundage (there might be some real purpose to this in a Dungeons and Dragons–type adventure). You might want to experiment with different weight limitations other than five objects when you actually start debugging your own adventure (five is an awfully small number in many situations, but again you have to be careful not to make it too easy). In any case, the value of W=2 here refers to the two objects our puppet robot is already carrying, the flashlight and the laser pistol, set in lines 2320 and 2330.

Line 2450 simply reminds us that all the doors are closed. We already discussed what the variable D was all about and how a value of 0 means closed door and a value of 1 means open door. As we said a moment ago, it is not necessary literally to set all these values, since they are all 0, but once again it is important to have a mention somewhere in the housekeeping of D(X) and what it is all about.

Line 2460 is similar to 2430 and 2450, again without the superfluous assignment of the 0 value. This line is self-explanatory and refers to the airlock door. And line 2470 RETURNs us back to line 70, whence we came.

## VARIABLE VARIABLES

One of the mysteries left unsolved in the last chapter was the strange case of the flexible R$. No matter which room we went into, part of the visual display included this strange string apparition. We said that R$ was a listing of the movable object contents of the given room, the CO$(X)s that corresponded to CO(X)s that were equal to R+100. But what were we talking about?

The problem we're trying to solve here is an important one. As our puppet

goes from room to room we want to display the contents of each room, but those contents are not all constant. The lead pentagon in room 4, for instance (also known as CO(2)/CO$(2)), is the first movable object our robot will run into, and no doubt the player manipulating the robot will want to pick that pentagon up. Originally we want the room display at the top to say SMALL LEAD PENTAGON along with the rest of the room description. And when our player commands TAKE PENTAGON, we want the words SMALL LEAD PENTAGON to disappear from the room display, since that item is not lying around in the room anymore but is now among the inventory of the puppet robot.

On the other hand, we know that our puppet is carrying what is listed as CO$(8), also known as SMALL FLASHLIGHT (we know this because we have already set CO(8) = 1). Let's say our player wanders into room 4 and sees the lead pentagon lying there. Before he takes the lead pentagon he wants to leave behind the small flashlight. So he types DROP FLASHLIGHT. When this command is executed we want the room display to list both the SMALL LEAD PENTAGON and the SMALL FLASHLIGHT. Then if he commands TAKE PENTAGON we want the pentagon to disappear while the flashlight remains. And in doing all this juggling we want the rest of the screen to remain unchanged. A tall order, perhaps, but we're ready to handle it.

Now you are going to begin to understand why we dimensioned one variable for all our movable objects and just what the values of those variables really do for us in a program. Let's look at it first and talk about it later:

***"Space Derelict!"***

```
13000 R$ = ""
13010 FOR N = 1 TO 16
13020 IF CO(N) = 100 + R THEN R$ = R$ + CO$
        (N)
13030 NEXT N
13040 RETURN
```

This little baby is the entire sum and substance of how we handle the itemizing of the movable objects in a room. We get to this subroutine in the program from line 50010, the beginning of the room-layout routine. We get to the room-layout routine in any of three ways: (1) our player enters a new room; (2) our player picks something up; (3) our player puts something down. A bit more explanation: Our player is in room 3 and he enters the command GO EAST. Our conversation with the computer segment first verifies that he actually can go east—that there's an exit in that direction and no locked doors to prohibit passage. If these criteria are met, the conversation routine resets the value of R to 4, which is the room east of 3. Then the program jumps to line 50000. With a new value of R—namely, R = 4—the program first travels over to 13000 to find out what items are lying around in the room, then trundles back again to the rest of the room-layout routine, branching from 50040 (the ON . . . GOTO command) to 50200, which provides the output to the screen that describes room 4, finally shuttling back to 3000 for more conversation with the player.

If our player commands TAKE PENTAGON, the program takes a similar tack. First the conversation with the computer segment verifies that the pentagon is in the particular room to be taken. If it is there, then the value of CO(2) is reset to 1, i.e., CO(2)=1. Then again the program jumps to line 50000. Once again the subroutine at 13000 is met, and this time, since CO(2) no longer equals 100+R, CO$(2) is not included in R$. Branch back to the ON . . . GOTO at 50040, but note that this time R has not been changed, so the branch goes to the same room. So when the output is displayed, the only things that change are the contents of the line that prints R$. Actually the entire output at the top of the screen will be cleared and reset, but the only noticeable difference will be in the contents of R$. Obviously the command DROP PENTAGON works in the opposite direction, verifying first that the robot has it to drop, then assigning a value to it of 100+R, then branching to 50000 to reset the top of the screen. A piece of cake when you know how to do it.

So let's look at the 13000 subroutine in detail. Line 13000 clears R$ and sets it at nothing—a null string, as they say in computer circles. This is how we can use the same R$ in every room; since we clear it right at the top, it's an empty variable when we start out, and we fill it up every time we come to this subroutine. A variable variable.

Lines 13010 and 13030 set up a FOR . . . NEXT loop of 16 values. We have 16 CO(X)s and CO$(X)s, which is where that number came from. You would obviously use the number of different CO(X) values you had put into your own adventure. If you're designing a treasure hunt, then you'll probably have twice as many movable objects floating around, and even that's being stingy. Then again, my philosophy is, the more you put in, the better.

Line 13020 is the meat of the matter. This line reads the value of CO(X) or (CO(N) in this case) and determines whether or not that value is equal to 100+R. It doesn't matter *which* R you happen to be talking about—R is whatever R was most recently set and therefore whichever room you happen to be in. If a movable object is in that room, then the string name of that item becomes part of R$.

Let's go back to room 4 to show you how that's done. The only item in room 4—i.e., the only CO(X) with a value of 104 (100+R where R=4)—is the SMALL PENTAGON, CO$(2). All together now, that means that CO(2)=104. With this in mind let's go through 13020. When N=1, CO(N)=CO(1). Since CO(1)<>104 (CO(1)=119—go back to your list of movable objects if you haven't already), we fall through this line and continue the loop. When N=2, CO(N)=CO(2), and CO(2) *does* equal 104. So we can continue on line 13020. Now R$, which previously equaled "" or zilch, now equals zilch plus CO(2). And "" + "SMALL LEAD PENTAGON." = "SMALL LEAD PENTAGON." (If you don't know exactly how this works, hang on, because we'll be talking about concatenation in a minute.)

For the rest of the N values CO(N)<>104, so that's all she wrote, and the total value of R$ remains SMALL LEAD PENTAGON. If there were more CO(N) values of 104, then their respective CO$(N)s would have been added to SMALL LEAD PENTAGON until all were accounted for.

The inventory subroutine is practically identical to the R$ subroutine. The only difference is that now we're evaluating whether or not the CO(X) value

of a given item is equal to 1. If CO(X) = 1, then we know that our puppet is carrying that particular CO(X). We get to the inventory subroutine from the conversation-with-the-computer segment when our player requests an inventory.

*"Space Derelict!"*

```
13100 INV$ = ""
13110 FOR N = 1 TO 16
13120 IF CO(N) = 1 THEN INV$ = INV$ + CO$(N
      )
13130 NEXT N
13140 PRINT "I AM CARRYING THE FOLLOWING:"
    : PRINT INV$
13150 GOTO 3000
```

As you can see, this is essentially a replay of the R$ routine, except this time we're setting the variable INV$ (that's INV as in inventory—unimaginative but effective). The only difference is the addition of line 13140, which is actually a display of INV$. Since an inventory has been asked for, this is how it is delivered. This information will show up at the bottom of the screen rather than at the top and simply will scroll up from below with the usual questions and answers.

This next thing is a very small point, and there are other ways of doing this, so I'll tell you how I did it and either you can do it my way or else you can enter what would have to be a very complicated formatting routine and do it your own way. I'm talking about line breaks. When you have a variable variable such as R$ you have to account somehow for the fact that your words might not break precisely at the end of a line, so when your adventure displays it it might look something like this:

SMALL LEAD PENTAGON. ODD PIE
CE OF METAL.

Some adventurers do find themselves getting hungry enough to look around for the odd pie lying about, but it does somewhat undercut the professionalism we're trying for to have such sloppy leftovers lying on the counter.

You could probably come up with a little assembly-language routine that would keep lines from breaking in midword, and you probably could even do it in Basic (perish the thought); but there's an easier way. Remember when I said to type in the list of movable objects *exactly* the way it was printed? If you did this you discovered that I had typed in a lot of blank spaces that seemed unnecessary. But in fact they were *quite* necessary. You see, every CO$(X) string is exactly 20 characters long (with the exception of the SMALL AREA OF WATER and the FIVE-LEGGED ROBOT, neither of which is actually movable). And as you know, the length of the screen output is either 40 or 80 characters long. So when R$ finally does get displayed, no matter how long it turns out to be, it is always divisible by 20 and therefore always breaks at a line ending. This may be the coward's way out of this predicament, but it works like a charm and it's highly recommended to my fellow cowards.

## CONCATENATION

Concatenation is the biggest word in the computer vocabulary and one of the simplest concepts. But unfortunately most computer writers wreak havoc with it, blowing it way out of all proportion to its simplicity. Since we're going to be concatenating up a storm in both of the game programs in this book, we might as well go over this well-trodden ground a bit ourselves.

Simply put, concatenation is the stringing together of two or more strings. If A$="FRED" and B$="J. MUGGS", and if we were to enter a command:

C$ = A$ + B$: PRINT C$

the computer would respond with:

FREDJ. MUGGS

which is the sum of the two strings A$ and B$. If we were to do it the other way around and say:

C$ = B$ + A$: PRINT C$

the computer would respond with:

J. MUGGSFRED

The computer doesn't care what's in your strings; when you concatenate them it adds them together, and you can do whatever you want with the results.

Let's go back to:

C$ = A$ + B$

where A$="FRED" and B$="J. MUGGS". This gives us a C$ of "FREDJ. MUGGS". The nice thing about string variables is that we can make them dance for us once we know how. We can turn strings into numbers, numbers into strings, read strings left to right and right to left or from the middle out—you name it, we can do it. And in programming games, sooner or later we will.

Let's assume that you are a good grammarian but not much of a trivia buff. Looking at C$ you realize right away that a space is missing between FRED and J. Let's add the missing space by using our dancing string commands. First, enter:

C$ = LEFT$ (C$,4) + " " + MID$ (C$,5)

Then enter:

PRINT C$

in order to get:

FRED J. MUGGS

If you know how that worked you can skip along to the next section. If you don't, settle in.

We can read any given string from right to left, from left to right, or from anywhere in the middle to the end. The commands we use to do this are LEFT$, RIGHT$, and MID$. The LEFT$ and RIGHT$ commands are the mirror images of each other: RIGHT$ counts off from the right, and LEFT$ counts off from the left. So the LEFT$(C$,4) that we used above took the string C$ and counted off four characters from the left. LEFT$(C$,3) would have given us "FRE", LEFT$(C$,2) would have given us "FR", and LEFT$(C$,8) would have given us "FREDJ. M". As long as the number is no greater than the total length of the string, we get the number of characters we ask for counting off from the left.

RIGHT$ works the same way from the right. RIGHT$(C$,4) would have given us "UGGS". As a matter of fact, we could have added our missing space in the following way:

C$ = LEFT$ (C$,4) + " " + RIGHT$ (C$,8)

since RIGHT$(C$,8)="J. MUGGS". Keep in mind that the space we added is, in effect, another string, a string one character long, that character being a space. This is *not* the same as a null string, which is no string at all, or a string of zero length.

You'll notice, though, that instead of originally using RIGHT$ to add the space I used the third string-reading command at our disposal, MID$. MID$ works from the middle, and it works in two ways, Let's look at C$ again as we've revised it:

C$ = "FRED J. MUGGS"

Let's say that this time we are true trivia buffs who can recognize that we have misnamed the chimpanzee who used to drive Dave Garroway crazy. That chimp's name was J. FRED MUGGS. Using MID$ we can adjust C$ accordingly:

C$ = MID$ (C$,6,3) + LEFT$ (C$,5) + RIGHT$ (C$,5)

If we were now to type:

PRINT C$

we would get:

J. FRED MUGGS

which is the correct and 100 percent bona fide name of the primate in question. How did we do that? MID$ reads a string from a specific given character for as many characters as we indicate in the string from left to right. So MID$(C$,6,3) reads C$ for three characters starting at the sixth character. Looking at the corrected C$, MID$(C$,4,4) would be equal to "FRED", which is the four characters staring at the fourth character. MID$(C$,10,5) would give us "MUGGS", which is the five characters beginning at the tenth character. But note also that MID$(C$,10) would *also* give us "MUGGS". When no second number is given to tell the computer how many characters to count off, the computer simply counts off to the end. This variation of MID$ has its uses too, and although it may look deceptively similar to the RIGHT$ command, there are times when the two will not be interchangeable, so you need both of them in your little bag of tricks.

Let's get into some muddier water. The computer also can turn numbers into strings and, on occasion, turn strings into numbers. And these too will be important concepts in game programming. Let's try a new formula:

H$ ="P23"

This formidable little item is quite similar to what we'll be working with later on, in our poker game. Let's further say that:

X = 2: Y = 3: Z = 4

Now we have three numeric variables, X, Y, and Z, and one string, H$. Let's say now that we want to change H$ and add to it the number 4 (the value of Z) at the end to get P234. There's an easy way to do that with the information at hand plus one new command, STR$:

H$ = H$ + STR$(Z)

STR$ takes the value of Z and translates it into a string. STR$ literally puts quotation marks around numbers. Since Z=4 then STR$(Z)="4". Now when we enter:

PRINT H$
we will get what we wanted, which is "H234". We can also work this the other way around. Let's say we want to find out the numeric value of part of a string. If we were to enter:
PRINT VAL (RIGHT$ (H$, 1) )
we would get the response:
4

VAL is the opposite of STR$. VAL turns strings into numbers, provided that string or part of a string is a number to begin with.
PRINT VAL (RIGHT$ (H$, 3) )
would give us:
234
as an answer. But:
PRINT VAL (LEFT$ (H$, 1) )
would get us into trouble, because LEFT$(H$,1) is the letter H, not a number, and therefore a valueless commodity as far as the VAL command is concerned.

This is not to say that the computer is ignorant of the ranks of letters in the alphabet, however. The Apple can compare alphabetical values if you so direct it. As far as the Apple is concerned, "A"<"B" and "B"<"C" and so forth down the line. Also, "BLACK"<"BOTTOM", "CLARK"<"KENT", again all the way down the line. So if you ever want to sort a list of names alphabetically, the Apple can do it for you.

There is one last string command worth knowing, LEN. LEN measures the length of a string.
PRINT LEN (C$)
would give us a response of:
13
because the character length of "J. FRED MUGGS" is 13. LEN can be handled just like any other number—that is, the command:
PRINT LEN (C$) * 3
would give you a response of:
39
which is 13 multiplied by 3. The VAL command can also be used in this way, as in:
PRINT VAL (RIGHT$ (H$, 4) * 3
which would give us an answer of 4 times 3, or 12.

The last important thing to mention about strings—and don't worry if you're still a little fuzzy on them because you'll see all these commands in action before this book is over—is that concatenation works only in one direction. There is no such thing as negative concatenation. Let's say that:
P$ = "MOTHER": Q$ = "HER"
If you were to try to get the word "MOT" in the following fashion, you would get a syntax error:
PRINT P$ − Q$
For some reason this seemingly logical handling of strings is unacceptable, even though it appears no more arcane then P$ + Q$. And there are times when a subtraction of strings would really fill the bill in what you are doing. But you're stuck without it. Never fear, though, because the commands you *do* have will do everything you need, provided you organize them correctly.

# SIMPLE FILE ROUTINES FOR SAVING GAMES

One of the nicest things about adventure games is the player's ability to save a game whenever he wants to. The next time he sits down and decides to attack it again he can indicate that he wants to play a saved game, and *voilà!* He is right back where he left off. This saves a lot of wear and tear roaming about in the same old dungeons for the umpteenth time when he's already scoured through them quite successfully, and it is surprisingly simple on the part of the programmer to include this feature in a game.

A game is saved on a disk via the agency of a text file. Now, anything on a disk is considered a file, but there are files and then there are files. Text files are very much like manila files in a filing cabinet: They are repositories for data, usually (although not necessarily) in an ordered, accessible format. Text files are used for such things as lists of addresses where the information is repetitive and in a precise format; each text file might contain one person's data, giving that person's name, address, telephone number, and occupation, always in the same order. Or each file could contain the data for 20 people in that same precise, rigid order. When it came to sorting these files you would always know where to look for whatever you wanted because it would always be in the same place: the name would always come first, then the address, etc. We're going to use this same concept for saving adventures.

The structure of text files is not complicated, but it is extremely rigid. Many programming tricks and techniques allow for somewhat sloppy procedures that nonetheless will work because of the forgiving nature of the programming language. But text files are not so lenient. If precise handling is not accorded them, they will come back and slap you down with the most amazing, unexpected results. Unless you're already fully versed and capable in the file department I would advise hitting the manuals one more time until you do feel comfortable with them. I'll do my best to explain them here for my own purposes, but they are much broader in scope that what we'll be doing, and they can solve an awful lot of programming problems for you if you know how to work them imaginatively and accurately.

Saving an adventure requires creating a *sequential* file, which is differentiated from a *random-access* file in exactly the way their names connote. A sequential file always is read from start to finish, whereas random-access files are open to more idiosyncratic approaches (and that is all we will be saying about random-access files in this book). A sequential file always is read from the first *field*—which is what we call a piece of information in a file—to the last field. And when we manipulate these files the computer in a sense really does read them à la manila folders.

The process is straightforward. The computer opens a file, reads it, and then closes it up and puts it away again. So let's say that a sequential file contains the following bits of information:

101
124
1
1
13
0

0
1
When we ask the computer to open up that figurative manila file, it reads these pieces of data in exactly this order (any of these pieces of data could also have been a string, but that is beside the point for our purposes), and now we can do whatever we want with them. But at the moment, they appear to be nothing more than a series of eight unrelated numbers. It is up to the program to make some sense of them.

The opposite of reading a file is writing a file. When we tell the computer to write a file it sets aside an imaginary new manila folder for us to insert whatever information we want into it. Then we tell the computer what that information is via PRINT commands. So if after opening a new file and directing the computer to write in it we say:

PRINT 101

the computer will write the number 101 as the first field in our new file. Following along in sequence, we could create the entire file of eight numbers listed above simply by repeating the PRINT command with the appropriate number after it (we'll get to the commands for the actual reading and writing of files shortly).

The computer also can take this one step further. Let's say we have a program where the variable X has a value of 101. If we wanted to write a file and enter the command:

PRINT X

the computer would print into the file the number 101, the value of X. Let's take it another step further and say we have a variable CO(3) in our program and the value of CO(3) is 101. If we command the computer to write a file and:

PRINT CO(3)

the computer would respond by writing the value of CO(3), which in this case is 101.

By now you should have guessed where this is all leading us. In our adventures we have a finite number of variables—R, the CO(X)s, the D(X)s, and so forth. We set them all and named them in the housekeeping routines, and along the way some of their values may have changed, but still they are the same in number, and more to the point, these variables and these variables alone are responsible for controlling the game. The values of these variables tell us what room we are in, what we are carrying, what doors are open and what ones are closed, what items are lying around where—literally everything there is to know about the game. So if we collect all these variables at any given point and save them, and then go back later and read them out, they will tell us exactly where we are within a game. Furthermore, by putting them into a text file we etch them onto a disk and store them away—that is, their values won't disappear when we turn off the machine. In other words, we can save a game in midstream and go back to it whenever we wish.

The trick, of course, is to write a file and read it in exactly the same sequence, thereby giving meaning to what would otherwise be a meaningless set of numbers. Before going into the mechanics of filing, let's take a look at the game-saving routines in "Space Derelict!"

```
30000 REM  SAVE GAME
30010 D$ = CHR$ (4)
30020 PRINT D$;"OPEN SDFILE"
30030 PRINT D$;"DELETE SDFILE"
30040 PRINT D$;"OPEN SDFILE"
30050 PRINT D$;"WRITE SDFILE"
30060 PRINT R
30070 FOR N = 1 TO 16
30080 PRINT CO(N)
30090 NEXT
30110 PRINT L
30120 PRINT W
30130 FOR N = 1 TO 17
30140 PRINT D(N)
30150 NEXT
30160 PRINT AL
30170 PRINT T
30180 PRINT D$;"CLOSE SDFILE"
30190 GOTO 3000
31000 REM  REOPEN SAVED GAME
31010 D$ = CHR$ (4)
31020 PRINT D$;"OPEN SDFILE"
31030 PRINT D$;"READ SDFILE"
31040 INPUT R
31050 FOR N = 1 TO 16
31060 INPUT CO(N)
31070 NEXT
31090 INPUT L
31100 INPUT W
31110 FOR N = 1 TO 17
31120 INPUT D(N)
31130 NEXT
31140 INPUT AL
31150 INPUT T
31160 PRINT D$;"CLOSE SDFILE"
31170 RETURN
```

Lines 30000–30190 contain the routine for saving a game, and lines 31000–31170 contain the routine for replaying a saved game. If you compare lines 30060–30170 to lines 31040–31150 you will find that they refer to exactly the same variables in exactly the same order. And as you also can see, they include all the variables with changeable values. They do not include, for instance, the string values of the CO$(X)s, because these values do not change. CO$(1) is ODD PIECE OF METAL and it *always* is ODD PIECE OF METAL regardless of where it is, and the value of the CO$(X)s will be set in the housekeeping routine at 2000 at the outset and need not be saved here. The only difference between the file that saves the game and the file that replays the game is the use of the PRINT statement vs. the use of the INPUT statement, which we'll explain now, as these are the mechanical aspects of filing.

Lines 30010 and 31010, the actual starts of both routines after the REMs,

set a new value for a new variable, the value being CHR$(4) and the variable being D$. The command CHR$(X) returns the ASCII value of X, as they say in the manuals. Each key on the keyboard has a different ASCII value (which are charted in all the manuals so I won't repeat them here). Further, the various combinations of operational keys also have ASCII values. The ASCII value of 4 refers to the combination CONTROL plus the letter D, CTRL-D. The CTRL-D function is the DOS operator for the opening and closing of files, for no other reason than that's the way it is. So if we want to open a file we "press" CTRL-D and then tell the computer OPEN FILE. For convenience's sake, a value of D$ is traditionally assigned to CTRL-D, which is what we've done in these two lines.

Lines 30020–30050 mechanically open the file. In effect what we are doing is concatenating a CTRL-D with a string containing another command such as OPEN or WRITE and also the name of the file. Line 30020 says "OPEN SDFILE"—SDFILE being the name we'll be giving to the file containing the "Space Derelict!" variables. This command figuratively reaches into the file cabinet and pulls out the file called SDFILE. If such a file is not in existence, then this command will create one. So OPEN does double duty either finding or creating the appropriate file.

Line 30030 oddly enough deletes all the information contained in SDFILE. While this may seem pointless, it is a good idea to erase the contents of a file before writing contents to that file so that you don't get left-over fields from earlier versions—this is part of the precision we talked about at the beginning of this section. Text files have their own whys and wherefores, and it is a good practice to follow them no matter what, although it is only for form in this example.

Line 30040 turns around and OPENs the SDFILE again, so having deleted the data in SDFILE in line 30030, we now have a fresh empty file re-created in line 30040. Line 30050 contains the command WRITE SDFILE, which means that from this point on all PRINT statements will send the data followed by the word PRINT into the SDFILE rather than the screen. The rest of the lines through 30170 make the actual delivery of the data. Line 30180 CLOSEs SDFILE (if we were to leave it open the havoc could be phenomenal), and 30190 sends us back to 3000, the conversation with the computer. A game can be saved at any point and does not necessarily mean that the game has ended, which is why the program branches in this fashion.

The replaying of a saved game works in much the same way, only in reverse. We OPEN SDFILE in line 31020, but this time we don't DELETE it because we want to see what's already there. This is done by the READ command in 31030. Now we want to take these saved values and set them into the memory of the computer, and we do this with the INPUT statements. When we're finished we CLOSE the file in line 31160 and then RETURN in line 31170 to where we started from (which happened to be line 90).

If your adventure has more or different variables than those here you can simply tack them onto this basic structure both for READing and WRITEing. The important thing is that the PRINT statements and the INPUT statements reflect each other in precise order: You don't want to input CO(X) when you think you're inputting values of D(X). Your program won't know the difference, but the player certainly will when he tries to proceed with the game.

And always adhere strictly to the various D$ commands or else be prepared for real disaster.

When passing your game along to others, it is a good idea to create a saved-game file and put it on the disk along with the game in case some wise guy tries to play a saved game his first time through (line 90). Simply start up a game and get yourself into room 1, where you SAVE GAME and that's the end of it. That way your wise guy won't bomb the program by trying to read a nonexistent file, and you won't give away anything he wouldn't have found out for himself anyway two minutes later.

## WINNING AND LOSING

The next two routines contain scenarios for winning and for losing the game. There is usually only one way to win any adventure: In a treasure hunt you have to collect all the treasure, and in a situational adventure you have to reach the end of the situation. In "Space Derelict!" the insertion of the filled box into the slot in room 24 signals winning the game, and as soon as this action is completed, the program branches to the winning scenario (which happens to be at line 17000). Any situational adventure will have a similar set of circumstances that signal success and a similar branch to the winner's circle.

Treasure-hunt adventures require some other apparatus to signal their successful completion. Usually you have to collect a certain number of treasures and then deposit them somewhere for safekeeping, generally some neutral room in the universe. An easy way to measure how the player is doing is to run a loop à la line 13000 every time he enters the command SCORE and count the number of treasures in the storage room. Let's say there are 20 treasures, and the storage room is number 32. This loop would figure out whether or not the player had won:

```
FOR X = 1 TO 20
IF CO(X) <> 132 THEN X = 20: Exit routine
NEXT X
GOTO 17000: REM WINNER'S CIRCLE
```

This routine accounts for each movable object. If any one of them is not in room 32, the loop is aborted, but if all 20 are there, you branch to the winner's circle.

In "Space Derelict!" the winning scenario looks like this:

***"Space Derelict!"***

```
17000 FOR N = 1 TO 20
17010 TEXT
17020 PRINT ""
    : REM  CTRL G
17030 NEXT
17040 INVERSE
17050 SPEED= 100
```

```
17060 PRINT "CONGRATULATIONS, PEOPLE OF THE
        SYSTEM   OF THE NINE PLANETS. YOU
       HAVE MANAGED TODEFUSE THE SHIP. IT
       SURPRISES US THAT   YOU WERE CAPABL
       E OF SUCH CLEVER          THINKING.
                                        "
17070 PRINT "
                  "
17080 PRINT "BUT BEWARE... THE KRENN WILL B
       E BACK    AGAIN SOME DAY. AND NEXT
       TIME IT WILL   NOT BE SO EASY FOR Y
       OU.                    "
17090 PRINT "
                  "
17100 PRINT "FAREWELL, PEOPLE OF THE SYSTEM
        OF THE   NINE PLANETS.
                         "
17110 PRINT "
                    END TRANSMISSION"
17120 END
```

There isn't much to winning other than satisfaction. Lines 17000–17030 set up a loop to ring the bell at CTRL-G 20 times—that way the player knows something noticeable is happening. Moreover, the TEXT command at 17010 negates all the window-setting POKEs from the room-layout routine, thereby ganging agley all the best-laid plans of mice and formatters. Now the player really knows something is happening. Then once again we slow down the display speed to 100 and congratulate the player, but it's just the sort of backhanded congratulation you would expect from the Krenn, with their threat of returning—as if they hadn't caused enough mischief already. Line 17120 ENDs the game once and for all.

A losing scenario assumes that the player has stepped into one of your deadlier traps from which there is no return. When this happens it is advisable not to be too snide about it; sooner or later the player *will* conquer your game, and he who laughs last . . . . Be nice to your audience; your player is doing you the supreme honor of spending his time with your game. So when the player loses, don't program a message such as:

PRINT "YOU LOSE, STUPID!"

That's just not good computer etiquette. But then again, the player did screw up, so you don't have to bend over backward to congratulate him on his intelligence, either.

Losing in "Space Derelict!" comes as a result of pressing a wrong door-control button, entering room 5 without deactivating the killer device, fighting a losing battle with the robot in room 23, plus one or two other miscellaneous screw-ups. This is what the losing scenario looks like:

***"Space Derelict!***

```
16000 POKE 34,0
16010 INVERSE
16020 FOR N = 1 TO 10
16030 PRINT "" + "END TRANSMISSION"
16040 PRINT
```

```
16050 FOR P = 1 TO 500
16060 NEXT
16070 NEXT
16080 NORMAL
16090 CLEAR
16100 PRINT
    : PRINT "IT LOOKS LIKE THE END OF THE S
      OLAR       SYSTEM. TOO BAD..."
    : PRINT
16110 INPUT "CARE TO TRY AGAIN?";C$
16120 IF C$ = "Y" OR C$ = "YES" THEN 10
16130 IF C$ = "N" OR C$ = "NO" THEN END
16140 GOTO 16110
```

Once again I took advantage of my fictional situation of a puppet robot, this time by printing a message that just reeks of computerese. Line 16000 serves the same purpose as the TEXT command in the winning scenario (I'm not being inconsistent: I'm just showing you another way to accomplish the same thing). The empty quotation marks in 16030 actually contain another CTRL-G, so once again we get a little session of bell ringing. It's the message that goes with it that adds that little computer touch—an inverse END TRANSMISSION printed over and over again. The first time the player sees it, it's quite a surprise; after that, it's quite a letdown. Lines 16050–16060 merely add a short pause.

Line 16120 gives the player an opportunity to try his luck again. If he does opt for another shot at it he's dealing from a cold deck, due to the CLEAR command in 16090. This line erases all the variables in the memory of the computer, so everything has to be reset all over again, which is why we go back to the beginning of the program, where once again we DIM this and that endlessly. Of course, once the player does get back to the beginning he has the option of taking up where he left off with a saved game if such is his pleasure, so although you're sending him back to square one in the program, he has his chance to jump ahead if he's stored any games-in-progress in his travels.

# 4

# CONVERSATION WITH THE COMPUTER

## THE NATURE OF THE BEAST

A player plays an adventure by sitting down at the computer and literally conversing with the program. The first step for the player is figuring out what language the game speaks. When he finds the right mode of communication, the right combination of words that the program can understand, the player can successfully get the program to respond to his commands. One of the joys of adventuring is that discovery of the extents and limitations of the game's vocabulary. Needless to say, the more extensive that vocabulary is, the more potentially rewarding it is to the player. And conversely, the more limited that vocabulary, the more frustrating the game can become. It is up to you as the game designer to develop as extensive a working vocabularly as you can for each game you create.

Adventures traditionally require one- or two-word inputs from the player, and although there are some games on the market that can handle sentences of more than two words, you're really asking for it if you try to emulate them at this stage of your career. Believe me, two words are enough to keep straight; more than that certainly stretches the capabilities of your average workingman's everyday Basic. The problem for the designer is squeezing in meaningful programming to analyze—or parse—these long sentences quickly and accurately. The work involved is a geometric progression of complexities as the player inputs get longer and longer, and more than two words can make the parsing a real nightmare. After a little practice you might be up to it, but for our purposes, two-word inputs will be plenty. Besides, in many respects the ability of a program to parse long sentences is merely window-dressing. It's quickest and easiest for the player to use the shortest inputs possible.

"Parse" is an interesting word. Parsing is breaking down a sentence into its components. Since in our adventures we are limiting ourselves to two-word sentences, the parsing becomes relatively automatic, since an imperative two-word sentence by definition contains exactly one verb and one noun, in that order. We're asking our player to enter commands—i.e., imperative sentences—so while there are conceivable complete two-word sentences not in the imperative mode, such as DOGS BARK, there is no earthly reason why any player would input such a sentence and expect any sort of response to it. An imperative sentence also can contain only one word, but that one word must be a verb (you wouldn't command someone to FRANKFURTER, for instance). Anyhow, the entire contents of the conversation-with-the-computer segment of the program consist of parsing the player input, analyzing it, and responding to it. Essentially you will be creating a working vocabulary, comparing what the player says to what you have in that vocabulary, then acting on the results.

There are certain words you will always use in every game, such as GO, TAKE, DROP, and LOOK. In fact, these particular four words provide the basic framework for your game vocabularies and will require some of the longest routines in the program. Other words will be more game-specific: If there is no body of water in the game universe, for instance, it is extremely unlikely that the player would come up with the command SWIM, and therefore it's relatively unnecessary for your program to accommodate that word. But if there *is* a body of water, your program should be ready to handle SWIM, WADE, FISH, DRINK, and anything else you can think of that remotely has to do with large amounts of liquid.

The keys here are flexibility and completeness. By this point in your game design you have set up all the factors of play, and you have incorporated all those factors into a universe waiting to be explored. But now you must prepare yourself to communicate to the explorer, and to do so in such a way as to be truly bend-over-backward accommodating. You have to prepare your program for *every* possible player input; more importantly, whenever that input is reasonable, even if the input is not "correct" vis-à-vis your game, you have to respond to it meaningfully. In effect you have to read the mind of every player who will ever sit down with your game.

It is extremely instructive to watch a complete novice attack an adventure. An experienced player becomes pretty good at getting instant communication with a new game, but a novice has really no idea of what's in store for him, so he'll try anything. And some of those tries aren't half bad. While some will be off the syntactical mark a bit (it takes a while to learn that sentences such as GO TREE or UNLIGHT LANTERN are actually meaningful), the vocabulary usage will be most illuminating. Not knowing the accepted modes of adventuring communication, the novice will instead try the more logical ones or else the most completely outrageous ones. This in fact becomes the true test of your adventure: When the player tries something quite logical in context, does your adventure respond just as logically? It should. And if the novice does something completely outrageous, does your adventure respond logically to that outrage, or is the response a non sequitur as silly as the input? If your adventure can't handle both the predictable and the unpredictable, it's not really ready for prime-time players.

One of the things we'll try to do in this chapter is create a catalog of good

vocabulary to include in adventures, but that catalog will by no means be complete. Every game you work on will have its own nooks and crannies where special words will be required that are unnecessary in other situations. In "Space Derelict!" for example, I use the words INSERT and MELT, both of which obviously fit very specific criteria here but that are not necessarily ever going to pop up again in other games. The point is, if there's nothing MELTable in the game universe, the player is never going to enter the command MELT MOUSE or whatever because it's just not going to make any sense and will therefore never occur to him. You have to be especially aware of those areas in your game where special, unusual vocabulary is needed and be prepared with as much of it as possible. And as a corollary to this, whenever possible you must be prepared with synonyms for your acceptable words. A moment ago I referred to LOOK, GO, TAKE, and DROP as the backbone words in the adventure vocabulary; so too are their synonyms EXAMINE, N (or E or W or S), GET, and LEAVE. For every verb there are probably two other similar verbs that you ought to include in your adventure. Of course, you can't throw the whole dictionary into the machine, so basic horse sense will limit you eventually, but the more common and likely variations all must go in to make the game really playable. The same holds true for nouns, but less strongly. Let's say, for instance, that a room is described as having a painting, and the player enters the command LOOK PICTURE. Is it necessary to respond to that command? Some would say yes, some would say no. I'm inclined against it myself in most cases, simply because there's enough work to do on the verb side of the statements without killing yourself on the noun side as well. There's no reason why a player can't describe an item using the words already on the screen in front of him. But if your room includes something called CRYSTAL WHALE, then it does make sense that your program be able to understand both the terms WHALE and CRYSTAL. But even here there's some debatability, and given my druthers I'd worry more about other aspects of vocabulary and let the player figure out that certain items have certain names, and if he gets it wrong the first time (by referring to the CRYSTAL, for instance), then he can get it right the second time (by referring to the WHALE), and he'll save us quite a bit of programming work by so doing.

No matter how much vocabulary you use, however, sooner or later your player will come up with something your program can't handle, and you have to be just as prepared with an adequate response here as you were when his command was workable. There are two general sorts of unacceptable player input, and you should respond to each appropriately. First, there is input in which the verb isn't included in your game vocabulary, and the player should be made aware of this in no uncertain terms. If a player enters SLURP SOUP, your program, which does not recognize the word SLURP, should respond with something like, WHAT DO YOU MEAN BY *SLURP*? This shows the player that SLURP is unacceptable but ignores SOUP entirely, leaving that issue for another day. On the other hand, let's say that you do include SLURP in your vocabulary, because slurping the soup is an important step in solving the puzzle, but your player, bumbling around, enters the command SLURP PIANO. Here we have a problem because we do know what slurping is, and we can't say we don't know what PIANO is because there indeed is a piano in your adventure. You can't recognize it one time

and not the next. To be on the safe side, when our program recognizes a verb but the noun following it is not one we are expecting, we simply vague it out for all wrong nouns with a response to the effect I CAN'T SLURP [WRONG NOUN]. This tells the player that the action requested is unacceptable but doesn't preclude use of the same vocabulary in other combinations.

The makeup of the conversation with the computer segment is not complex, although the segment itself is long (as a rule, the longer this segment the better your game because it can respond to that much more vocabulary). The segment begins by getting the input from the player; then it analyzes that input to determine the length of the first word (or, if it is a one-word command, the length of that one word). Then the program scuttles off to an appropriate section consisting of words of given lengths: All the one-letter first-word commands are grouped together, as are the two-letter commands, the three-letter commands, etc. The computer compares the player input to the words in the appropriate group, and if a match is found, then it follows the instructions at that match and the game will progress accordingly. If no match is found by the end of the group, then the program will respond that the first word of the player input is not one that the computer understands, and the player will be prompted for a new input.

As I've pointed out before (and which is now quite obvious since we've pretty much seen all of the rest of the program), most of the real business of the game is handled in the conversation segment. If a player TAKEs or LEAVEs a movable object, this is where it is done. If a player wants to move the puppet, this segment does the work. If the player wants to look at something, or touch it, or kick it, or eat it, or whatever, this is where the action is. Fortunately, though, none of this is very complicated in programming terms, so while the sheer length of what we'll be doing here may be daunting (even the simplest adventure can eat up 30K of memory without batting an eye), the process of doing it is very simple, so don't be put off by the vastness of it all.

## WORDS AND RESPONSES

Anyone who plays a few adventures by different designers soon finds that the commands that work for one don't always work for another. One game will allow you to GO NORTH and to GO DOOR, while another will accept any directional command but will spit the more specific GOs right back at you, requesting instead a direction. Some games will accept INV and INVENTORY and TAKE INVENTORY as well as such typing-creative variations as INVETNORY and TAKE INVNET (the program management of these niceties is simple and you'll see some of it in "Space Derelict!"). As a rule, the more adaptable your game is, the more fun for the player, but you can run yourself ragged trying to accommodate all the typing errors that in many cases ought simply to be shipped back to the player. You'll have to make up your own mind how adaptable you want to be, because it's your time that is going to be spent hacking at the computer.

We're going to list a bunch of words here, and while this list is in no way inclusive of all the vocabulary you can use in a game, it is a good shopping list to start with when you program your own conversation segments. Not all

of these words are included in "Space Derelict!" by any means; those that are are marked by an asterisk. Also, not all of these words demand full programming space in the segment, as some are simply synonyms for others. TAKE and GET, for instance, are interchangeable, and the required actions are accounted for only at one stand in the program; if the other word is input by the player, the program simply substitutes the word it understands and acts accordingly (we'll see how that's done later).

The words below are grouped by length, as they are in the actual program.

One-letter:

N*
S*
E*
W*
U
D

(All of the above are, of course, directions.)

Two-letter:

GO*

Three-letter

RUN*
GET*
SAY*
EAT
BUY
FLY
SIT
TRY
CUT*
CRY*
TIE*

Four-letter:

READ*
TAKE*
SAVE*
QUIT*
SWIM
HIDE
GIVE
SING
KISS
PLAY
KICK*
HELP*

KILL*
OPEN*
LOOK*
DROP*
FIND*
MELT*
LOCK*
MAKE
PUSH*

Five-letter:

CLIMB*
POINT
LIGHT*
BUILD
LEAVE*
SCORE*
SHOOT*
START
DRINK
BRIBE
WHERE*
SMOKE
FRISK
PRESS*
ENTER*
CLICK*
TOUCH*
THROW*
LASSO*
BLAST*
THINK*
BREAK
SMELL

Six-letter:

UNLOCK*
TICKLE
SEARCH
INSERT*
REMOVE*

Seven-letter:

WRESTLE
UNLIGHT*
EXAMINE*

Eight-letter:

DESCRIBE*

Nine-letter:

INVENTORY*

As you can see, we've got quite a range here, and even as you read them you probably thought of a dozen more that would have fit in nicely. Some of these are quite specific to precise circumstances in a given game universe, but others are general enough to merit inclusion in every adventure. This list is merely a starting point for your own creativity; don't take it as gospel because it's only one man's list of words I like, and you can amend, adjust, and expand it as you see fit.

The responses the computer displays on the video screen must be specific to your game and the situation in which the player finds himself at the moment. In "Space Derelict!" since these responses are coming from a robot there is a certain deliberate robotic woodenness to the wording that wouldn't work for other games. You might want to go back now to the listings from 2020 to 2250, which we covered in the previous chapter, to follow the way the responses were handled in "Space Derelict!" You'll recall that our catchall array for answers from the computer was the string variable A$(X).

A$(1) answers the situation mentioned earlier in this chapter when the input is somewhat recognizable but the combination of the words doesn't make sense in the program. Let's say the player commands CLIMB HOLOGRAM, for instance. The program will recognize CLIMB, but nothing in the CLIMB section will account for going up the hologram, so the program responds in such a way as not to discourage the player from climbing but to tell him that at the moment he's up the wrong tree. A$(2) is the other half of the coin, where the first word of the command is not recognized by the program. When that word isn't found, the program prints A$(2) *plus* the word in question (although that isn't obvious just by looking at line 2040—please stay tuned for specifics), and that way the player knows not to try that word again. Also, if the player has input a typo, this way he will see delivered back to him exactly what he typed in, so he'll know the mistake was mechanical and not metaphysical.

A$(3) is so optional that I don't think it's ever used in "Space Derelict!" even though it's set so nicely right at the beginning. I believe that every time the computer accepts a command it should have a default response of OKAY, but it's a hell of a lot easier to type OKAY than it is A$(3) all the time. Suit yourself with this one; you'll find "Space Derelict!" loaded with OKAYs, in any case.

A$(4) and A$(5) are self-explanatory. A$(4) comes up when a player tries to go north or wherever and there's no north to go to, while A$(5) reminds him that the door ahead is locked.

A$(6) takes care of situations where the player is trying to do something it's okay to do but not at the moment. CLIMB ROPE, for instance, is perfectly acceptable in "Space Derelict!" as there is a rope in room 22 just ripe for the climbing. But if our player is in room 4, where there is no rope,

we have to handle CLIMB ROPE in such a way as merely to postpone it, not to cancel it entirely. A$(6) does this for us.

A$(7) is the response to commands to somehow manipulate an object not presently available and is much like A$(6) in that it keeps the player hopeful that whatever he's looking for will soon turn up.

A$(8) answers those commands where the player is trying to get the program to solve the problem for him. For instance, let's say the player comes upon the first locked door in room 3 and simply commands UNLOCK DOOR. Of course he wants to unlock the door, but he's got to discover the means of so doing first, hence the response A$(8), which puts him in his place and lets him know that he's going to have to do all the work himself.

A$(9) is much like A$(1) but is a little more precise and leading. We'll see how it's used when we get further into the program.

A$(10) through A$(20) are descriptions of various items that will be supplied if the player requests them. They are loaded here rather than in the conversation section simply to save space on the programming lines where they occur. Any one programming line can contain only 255 characters, and there is a danger with some of those lines of exceeding that limitation.

A$(21) is the response to commands that just don't work, like trying to click the door opener at a door that's already open, or at a door that opens to a different command.

A$(22) and A$(23) are two more descriptions, A$(22) being of the non-descriptive variety, the response to describe something that doesn't require more description, and A$(23) obviously being the robot's laser.

There are other responses from the program buried away in the conversation segment, occurring at those one-of-a-kind situations where it was easier to include it there than to come back and amend the A$(X) variables. But you can see here a general pattern of using variables for the more frequently necessary responses—no can do, nothing happens, wrong direction, etc.—that you should include in your own adventures to save yourself programming time when you dive into the actual conversation segment.

## THE MECHANICS OF THE CONVERSATION SEGMENT

The first step in the conversation segment is to break down the player input into a form recognizable to the program. Since the segment is arranged according to the length of the first word of the input (or only word, if it's a one-word input), the first thing we do is get the length of that first word. We might as well go now to the program to see how it's done.

*"Space Derelict!"*

```
3000  REM  Q&A
3010  IF R = 23 THEN GOSUB 40000
3020  INPUT "-->WHAT SHOULD I DO?";C$
3030  IF C$ = "" THEN PRINT A$(1)
    : GOTO 3000
3040  IF LEN (C$) = 1 THEN 3140
3050  IF MID$ (C$,2,1) = " " THEN PRINT A$(
    : GOTO 3000
      2) + LEFT$ (C$,1)
```

```
3060  IF LEN (C$) = 2 OR MID$ (C$,3,1) = "
        " THEN 4150
3070  IF LEN (C$) = 3 OR MID$ (C$,4,1) = "
        " THEN 4670
3080  IF LEN (C$) = 4 OR MID$ (C$,5,1) = "
        " THEN 5070
3090  IF LEN (C$) = 5 OR MID$ (C$,6,1) = "
        " THEN 7210
3100  IF LEN (C$) = 6 OR MID$ (C$,7,1) = "
        " THEN 7910
3110  IF LEN (C$) = 7 OR MID$ (C$,8,1) = "
        " THEN 8100
3120  IF LEN (C$) = 8 OR MID$ (C$,9,1) = "
        " THEN 8190
3130  IF LEN (C$) = 9 OR MID$ (C$,10,1)
        = " " THEN 8240
3135  PRINT A$(2) + C$
    : GOTO 3000
```

Line 3000 reminds us where in the program we are, and line 3010 is an exception to all adventure-design rules, a piece of miscellany that doesn't fit anywhere else (we'll describe miscellany in detail at the end of this chapter). As you can see, this line comes into play only if the player is in room 23, so it certainly doesn't affect most of the game and can easily be ignored for the time being.

Line 3020 gets the player input, which will be the string C$. You can rephrase your question in this line any way you want if you're not quite satisfied with—>WHAT SHOULD I DO? but keep in mind that as programmed the INPUT statement will print the player's input on the same line as these words as it is typed in, so your prompt words should be kept short for the player's input to fit all on one line.

Lines 3030 through 3130 measure the length of the first word of the input, then route the program to the appropriate area to handle words of that length. All possible lengths, including null strings of zero length, are accounted for. Line 3030 is the null-string handler; the only time a string will come in null is when the player hits the Return button before he puts in his C$, so A$(1) simply tells him to get his act together, and then the program routes back to 3000. Keep in mind that not only does the player see A$(1) printed on the screen, but also the—>WHAT SHOULD I DO? prompt again as well printed below it.

Line 3040 accounts for one-letter-long inputs, routing the program to line 3140, the beginning of the one-letter group. Line 3050 is in effect a typo handler. It reads C$ and, if C$ is longer than one letter (the program would have already branched from 3040 if it was one letter), this line looks for a space as the second character. We're looking for input here along the lines of G DEVICE or I ICE—meaningless stuff that needs to be sent back to the player, with the A$(2) explanation concatenated to the screwy one-letter first-word input that caused the problem, in this case LEFT$(C$,1). If in fact the input had been G DEVICE, then the player would have seen:

I'M SORRY, BUT I CANNOT RESPOND TO THE COMMAND
G
—>WHAT SHOULD I DO?

A$(2) is designed so that when it tells the player what it can't respond to, the line break falls in such a way that C$ is not broken too infelicitously. You'll notice that a similar error catchall does not follow the subsequent length-checking lines. Rest assured that this function is carried out within the programming for groups of these lengths.

Lines 3060 through 3130 are virtually identical, so we'll analyze just one of them. The first thing 3060 does is figure if C$ is two characters long. If that is the case, processing will continue along this line—i.e., a branch to 4150. The OR provision figures whether the third character of C$ is a space, which would mean, of course, that the first word of C$ is two letters long. Now we have accounted for either a one-word input two characters long or an input of any length where the first space found is the third character, meaning again that the first word is two characters long. Keep in mind that each successive line of the program moves the measurement of that space character one character along: Line 3050 looks for the space at character number 2, 3060 looks for it at character number 3, and so on, so if that space criterion is not met, the processing simply continues one line further down. Each of these lines from 3060 through 3130 provides a branch to the appropriate word-length group for lengths from two to nine characters. Line 3135 is our default, accounting for first word-length inputs of 10 characters or more; once again, A$(2) plus this time all of C$, whatever its length, are returned to the player as unacceptable. And we've now accounted for every possible length and variation of C$. We haven't done much with it yet, but at least we know what it is and, if we're going to act upon it, how long it is. This length information will remain important as we continue processing the data contained in C$.

## MOVING AROUND

There's a lot of programming in the conversation-with-the-computer segment, and much of it is repetitious. While we will go through all of it in this chapter, there is really no point in explaining every single line when we've covered the gist of that line earlier. Take moving from room to room, for instance. The programming lines covering this extend from 3140 through 4140, chock full of the same things over and over and over again. So there will be a little less detail as we progress through this section, but we will cover everything as clearly as possible.

Movement from room to room will stem from the player inputting either a direction such as N or E or the more complete input GO NORTH or GO EAST. The section of the program covering this movement is broken down into each of the four possible directions, so if the player inputs W then the computer processes the west section to act upon that input. Two things need to be figured out by the program in order to act. First, it must determine whether the puppet can in fact move in that direction. If the player wants to go west but there is no west exit, the program must come back to the player

with A$(4), I CANNOT GO IN THAT DIRECTION, and then reroute back to line 3000. Second, if there *is* a west exit, then the program must determine whether the door between the player and the room to the west is open or shut. This is done by checking the value of the appropriate D(X). If the door is shut, then the program must return to the player with A$(5), THERE IS A LOCKED DOOR IN THAT DIRECTION, and a reroute back to 3000. If it's open, then the program must move the player into that west room. The actual movement is handled very simply through changing the value of the variable R. If the player is in room 4, for instance, and he enters the GO WEST command, the program reassigns a value of 3 to the variable R and then routes through the room-setting routine at 50000, finally coming back to 3000 from the direction at the end of the 50000 routine. All very simple, but the program must evaluate all these variations for each and every possible room.

Sitting down to program this section is not hard, just time-consuming. Start with any direction ("Space Derelict!" starts with north), then run through the three procedures for each room. The three procedures again are: is/isn't exit in that direction; door locked/not locked; stay put and back to 3000/make move then back to 3000. Let's look at these lines in "Space Derelict!" and cover the important points.

***"Space Derelict!"***

```
3140  IF C$ = "S" THEN 3470
3150  IF C$ = "N" THEN 3190
3160  IF C$ = "W" THEN 3810
3170  IF C$ = "E" THEN 3980
3180  PRINT A$(1)
    : GOTO 3000
3190  IF R < 3 THEN 3180
3200  IF R < 5 THEN PRINT A$(4)
    : GOTO 3000
3210  IF R = 5 THEN R = 11
    : PRINT "OKAY."
    : GOTO 50000
3220  IF R = 6 AND D(1) = 0 THEN PRINT A$(5
      )
    : GOTO 3000
3230  IF R = 6 AND D(1) = 1 THEN R = 8
    : PRINT "OKAY."
    : GOTO 50000
3240  IF R = 7 AND D(2) = 1 THEN R = 6
    : PRINT "OKAY."
    : GOTO 50000
3250  IF R = 7 AND D(2) = 0 THEN PRINT A$(5
      )
    : GOTO 3000
3260  IF R = 8 THEN PRINT "OKAY."
    : R = 9
    : GOTO 50000
3270  IF R = 9 THEN PRINT A$(4)
    : GOTO 3000
```

```
3280  IF R = 10 AND D(4) = 1 THEN R = 22
    : PRINT "OKAY."
    : GOTO 50000
3290  IF R = 10 THEN PRINT A$(5)
    : GOTO 3000
3300  IF R = 11 THEN R = 4
    : PRINT "OKAY."
    : GOTO 50000
3310  IF R = 12 AND D(3) = 1 THEN PRINT "OK
      AY."
    : R = 10
    : GOTO 50000
3320  IF R = 12 AND D(3) = 0 THEN PRINT A$(
      5)
    : GOTO 3000
3330  IF R < 18 THEN PRINT A$(4)
    : GOTO 3000
3340  IF R = 18 AND D(5) = 1 THEN R = 21
    : PRINT "OKAY."
    : GOTO 50000
3350  IF R = 18 AND D(5) = 0 THEN PRINT A$(
      5)
    : GOTO 3000
3360  IF R = 19 AND D(6) = 1 THEN R = 18
    : PRINT "OKAY."
    : GOTO 50000
3370  IF R = 19 AND D(6) = 0 THEN PRINT A$(
      5)
    : GOTO 3000
3380  IF R = 20 THEN PRINT A$(4)
    : GOTO 3000
3390  IF R = 21 AND D(7) = 1 THEN R = 23
    : PRINT "OKAY."
    : GOTO 50000
3400  IF R = 21 AND D(7) = 0 THEN PRINT A$(
      5)
    : GOTO 3000
3410  IF R = 22 AND D(8) = 0 THEN PRINT A$(
      5)
    : GOTO 3000
3420  IF R = 22 AND D(8) = 1 THEN PRINT "OK
      AY."
    : R = 23
    : GOTO 50000
3430  IF R = 23 AND D(9) = 1 THEN R = 24
    : PRINT "OKAY."
    : GOTO 50000
3440  IF R = 23 AND D(9) = 0 THEN PRINT A$(
      5)
    : GOTO 3000
3450  IF R = 24 THEN PRINT A$(4)
    : GOTO 3000
3460  GOTO 3180
3470  IF R < 3 THEN PRINT A$(1)
    : GOTO 3000
```

```
3480  IF R = 3 AND D(10) = 1 THEN R = 2
    : PRINT "OKAY."
    : GOTO 50000
3490  IF R = 3 THEN PRINT A$(5)
    : GOTO 3000
3500  IF R = 4 THEN R = 11
    : PRINT "OKAY."
    : GOTO 50000
3510  IF R = 11 AND D(11) = 1 THEN R = 5
    : PRINT "OKAY."
    : GOTO 50000
3520  IF R = 11 THEN 16000
3530  IF R = 5 THEN PRINT A$(4)
    : GOTO 3000
3540  IF R = 6 AND D(2) = 1 THEN R = 7
    : PRINT "OKAY."
    : GOTO 50000
3550  IF R = 6 THEN PRINT A$(5)
    : GOTO 3000
3560  IF R = 7 THEN PRINT A$(4)
    : GOTO 3000
3570  IF R = 8 AND D(1) = 1 THEN R = 6
    : PRINT "OKAY."
    : GOTO 50000
3580  IF R = 8 THEN PRINT A$(5)
    : GOTO 3000
3590  IF R = 9 THEN R = 8
    : PRINT "OKAY."
    : GOTO 50000
3600  IF R = 10 AND D(3) = 1 THEN PRINT "OK
      AY."
    : R = 12
    : GOTO 50000
3610  IF R = 10 THEN PRINT A$(5)
    : GOTO 3000
3620  IF R < 18 THEN PRINT A$(4)
    : GOTO 3000
3630  IF R = 18 AND D(6) = 1 THEN PRINT "OK
      AY."
    : R = 19
    : GOTO 50000
3640  IF R = 18 THEN PRINT A$(5)
    : GOTO 3000
3650  IF R = 19 OR R = 20 THEN PRINT A$(4)
    : GOTO 3000
3660  IF R = 21 AND D(5) = 1 THEN PRINT "OK
      AY."
    : R = 18
    : GOTO 50000
3670  IF R = 21 THEN PRINT A$(5)
    : GOTO 3000
3680  IF R = 22 AND D(4) = 1 THEN R = 10
    : PRINT "OKAY."
    : GOTO 50000
3690  IF R = 22 AND D(4) = 0 THEN PRINT A$(
      5)
    : GOTO 3000
```

```
3700  IF R < > 23 THEN 3790
3710  INPUT "WEST OR EAST?";C$
3720  IF C$ = "W" OR C$ = "WEST" THEN 3750
3730  IF C$ = "E" OR C$ = "EAST" THEN 3770
3740  GOTO 3000
3750  IF D(8) = 1 THEN R = 22
    : PRINT "OKAY."
    : GOTO 50000
3760  GOTO 3780
3770  IF D(7) = 1 THEN R = 21
    : PRINT "OKAY."
    : GOTO 50000
3780  PRINT A$(5)
    : GOTO 3000
3790  IF R = 24 THEN R = 23
    : PRINT "OKAY."
    : GOTO 50000
3800  GOTO 3180
3810  IF R < 3 THEN PRINT A$(1)
    : GOTO 3000
3820  IF R = 3 AND D(12) = 1 THEN R = 6
    : PRINT "OKAY."
    : GOTO 50000
3830  IF R = 3 THEN PRINT A$(5)
    : GOTO 3000
3840  IF R = 4 THEN R = 3
    : PRINT "OKAY."
    : GOTO 50000
3850  IF R < 9 THEN PRINT A$(4)
    : GOTO 3000
3860  IF R = 9 THEN R = 10
    : PRINT "OKAY."
    : GOTO 50000
3870  IF R = 10 AND D(13) = 0 THEN
        PRINT A$(5)
    : GOTO 3000
3880  IF R < 14 THEN PRINT A$(4)
    : GOTO 3000
3890  IF R = 14 AND D(14) = 1 THEN R = 15
    : PRINT "OKAY."
    : GOTO 50000
3900  IF R = 16 THEN R = 9
    : PRINT "OKAY."
    : GOTO 50000
3910  IF R = 14 THEN PRINT A$(5)
    : GOTO 3000
3920  IF R = 17 AND D(17) = 1 THEN R = 15
    : PRINT "OKAY."
    : GOTO 50000
3930  IF R = 17 THEN PRINT A$(5)
    : GOTO 3000
3940  IF R = 18 THEN R = 17
    : PRINT "OKAY."
    : GOTO 50000
3950  IF R = 19 THEN PRINT A$(4)
    : GOTO 3000
```

```
3960  IF R = 20 THEN R = 18
    : PRINT "OKAY."
    : GOTO 50000
3970  GOTO 3180
3980  IF R < 3 THEN PRINT A$(1)
    : GOTO 3000
3990  IF R = 3 THEN R = 4
    : PRINT "OKAY."
    : GOTO 50000
4000  IF R < 6 THEN PRINT A$(4)
    : GOTO 3000
4010  IF R = 6 AND D(12) = 1 THEN R = 3
    : PRINT "OKAY."
    : GOTO 50000
4020  IF R = 7 OR R = 8 THEN PRINT A$(4)
    : GOTO 3000
4030  IF R = 9 THEN R = 16
    : PRINT "OKAY."
    : GOTO 50000
4040  IF R = 10 THEN R = 9
    : PRINT "OKAY."
    : GOTO 50000
4050  IF R = 11 OR R = 12 THEN PRINT A$(4)
    : GOTO 3000
4060  IF R = 13 AND D(15) = 1 THEN
        PRINT "OKAY."
    : R = 15
    : GOTO 50000
4070  IF R = 13 THEN PRINT A$(5)
    : GOTO 3000
4080  IF R = 14 THEN PRINT A$(4)
    : GOTO 3000
4090  IF R = 16 AND D(16) = 1 THEN R = 15
    : PRINT "OKAY."
    : GOTO 50000
4100  IF R = 16 THEN PRINT A$(5)
    : GOTO 3000
4110  IF R = 17 THEN R = 18
    : PRINT "OKAY."
    : GOTO 50000
4120  IF R = 18 THEN R = 20
    : PRINT "OKAY."
    : GOTO 50000
4130  IF R > 18 THEN PRINT A$(4)
    : GOTO 3000
4140  PRINT A$(1)
    : GOTO 3000
```

Lines 3140 through 3180 evaluate the contents of a one-letter C$ input, branching to the appropriate direction in the rest of this section. S, N, W, and E are accounted for with branches, while any other one-letter input gets the A$(1) raspberry and an express train back to 3000.

So let's imagine we're programming now, and we're going to evaluate all the rooms in response to the N command. We'll be starting here at 3190. First we take room 1. If you look at the description of room 1 in the room

routine at 50000, you'll see that no exit information is given. The only way the player can proceed from room 1 to room 2 is with an ENTER AIRLOCK or GO AIRLOCK. The same is true for room 2, where ENTER SHIP or GO SHIP must be used, so if R<3 then we return the response of A$(1), unacceptable instruction. So we move along to room 3. Here we have regular exits, but north is not one of them. As it turns out, the same is true for room 4; hence line 3200 returns A$(4), can't go in that direction, for both values. Room 5 does have a north exit and no door, so when a player is in room 5 he can unconditionally go north. Looking at the map, we see that north of room 5 is room 11; hence line 3210, which changes the value of R and then branches to 50000. You'll notice too the response OKAY, just because it's polite to answer people when they talk to you. You'll also notice that this proves the author's reluctance to type in A$(3) when a simple OKAY will suffice.

Room 6 gives us our final possibility, as there is a door to the north of this room (which happens to be D(1)). If D(1)=0, which means a closed door, then we print A$(5), advising our player of the situation and branching back to 3000. This is done in 3220. Line 3230 handles the other possibility and moves the player to room 8 accordingly. Note that line 3230 need not include the AND D(1) = 1, because if it doesn't equal 0 than it *has* to equal 1. This concept of logical exclusion is simple: If there are only two ways of doing something and you've just eliminated the first one, then you have only the second one left to contend with. Lines 3280–3290 show you logical exclusion at work in actual programming statements. Line 3280 covers the possibility of our being in room 10 with the door open. The only other possible door condition if we are in room 10 has to be door closed, and the only way line 3290 will be processed if R does equal 10 is if D(4) is equal to 0. Therefore all we have to write in 3290 is IF R = 10, since the door condition is implicit in our having gotten to this line in the first place. As you can see, logical exclusion certainly can help streamline our programming, but it suffers at times from being somewhat unclear to the person coming at our program with less than our complete understanding of it. For the sake of clarity I've kept down the use of logical exclusion for the time being, but you'll find it snuck in shortly because you should get used to the idea sooner or later. In the poker program we will use it all the time, so we'll ease into it now in "Space Derelict!"

The rest of the lines through 3450 cover all the rooms going north, doing exactly the same things we've just been doing, so I won't go into them any further. Line 3460 is a simple redundancy that can easily be eliminated, but I've ended each of the four directional segments with a similar line just to be on the safe side. It doesn't hurt, and it helps me sleep better at night just knowing it's there. If there were a logical error in one of these segments these catches would make that unequivocally apparent at the debugging stage, which makes for easier debugging than leaving it all to chance.

The rest of the lines up through 4140 simply recapitulate the ones above, but for the other three directions, and with one exception there's simply no point in going into them again here; we've got programming of a lot more interesting nature ahead of us. But there is one variation that needs to be pointed out. You'll remember that room 5 is a trap and that entering room 5 without first "opening" the imaginary door that deactivates the lethal device

in the ceiling will kill the puppet robot. So we account for this in lines 3510 and 3520. If the door is open, hunky-dory, and we enter the room as usual. But if D(11) = 0 then we branch to line 16000, the losing routine. The player does get to look longingly into room 5 from room 11, but he'll never get in room 5 until he deactivates the deadly device. And another one bites the dust.

## GOING OTHER PLACES—TWO LETTER WORDS

You have a decision to make at the outset of your designing: Is your game only going to respond to directional commands, or can the player get to a specific place or item by such commands as GO DOOR and GO TABLE? I have seen games designed both ways, and quite frankly I prefer the latter as a player. When you see something sitting right in front of you in a room—a chair, for instance—it is mightily frustrating to type in GO CHAIR only to get the response I NEED A DIRECTION. Not that it's so hard to map this sort of thing as a player, but it is a pain in the neck. It is a tad harder on the programming side, obviously, because not only do you need to travel from room to room in response to directional commands, but also from place to place within rooms. If you prefer the directional-command approach, all you have to do is adjust the section we've just gone through with an eye on some of the things we'll be discussing in this section.

We're going to introduce a new twist to our analysis of C$ this time out. We already figured out the length of the first word back at the outset of the conversation-with-the-computer routine. Now we're going to refine that one step further by giving a name and a definition to all the rest of C$. We know that if we have branched to line 4150 we are dealing with a C$ that consists of (1) a two-letter word followed by (2) a space. What follows is still a mystery, but at least if we quantify it we can hold it up to the light a little better. So we're going to enter the following line (plus lines similar to it for each consecutive word-length grouping):

S2$ = MID$ (C$,3)

This gives us the new string S2$, which consists of all of C$ *starting with the space*. This distinction is very important both for typographical reasons (we'll always have to include the space in our programming lines, as you'll soon see) and for LEN length reasons, due to various logical exclusions we will encounter (usually at the beginning of length groupings). You'll see what I'm talking about as we progress; the important thing is that we now have something other than the first word of C$ to analyze. And analyze it we will. Keep in mind that the use of MID$ here takes C$ from character number 3 all the way to the end of the string, regardless of its length. The reason that we don't define S2$ as C$(MID$,4)—i.e., starting at the first letter after the space—is that C$ could have a LEN of only 3 as defined in line 3060, and therefore C$(MID$,4) would crash the program.

We should go now to the relevant programming lines in "Space Derelict!" for a more specific discussion.

```
4150  REM  2 LETTER 1ST WORDS
4160  IF C$ = "GO" THEN PRINT A$(1)
    : GOTO 3000
4170  IF LEFT$ (C$,2) < > "GO" THEN
        PRINT A$(2) + C$
    : GOTO 3000
4180  S2$ = MID$ (C$,3)
4190  IF C$ = "GO NORTH" THEN 3190
4200  IF C$ = "GO SOUTH" THEN 3470
4210  IF C$ = "GO WEST" THEN 3810
4220  IF C$ = "GO EAST" THEN 3980
4230  IF R < > 15 THEN 4340
4240  IF C$ = "GO EAST(N)" AND D(17) = 1 THEN
        R = 17
    : PRINT "OKAY."
    : GOTO 50000
4250  IF C$ = "GO EAST(N)" THEN PRINT A$(5)

    : GOTO 3000
4260  IF C$ = "GO EAST(S)" AND D(14) = 1 THEN
        R = 14
    : PRINT "OKAY."
    : GOTO 50000
4270  IF C$ = "GO EAST(S)" THEN PRINT A$(5)

    : GOTO 3000
4280  IF C$ = "GO WEST(N)" AND D(16) = 1 THEN
        R = 16
    : PRINT "OKAY."
    : GOTO 50000
4290  IF C$ = "GO WEST(N)" THEN PRINT A$(5)

    : GOTO 3000
4300  IF C$ = "GO WEST(S)" AND D(15) = 1 THEN
        R = 13
    : PRINT "OKAY."
    : GOTO 50000
4310  IF C$ = "GO WEST(S)" THEN PRINT A$(5)

    : GOTO 3000
4320  IF C$ = "GO PANEL" THEN PRINT "THERE
        ARE NUMEROUS CONTROLS HERE, BUT   N
        ONE OF THEM SEEM TO BE OPERATIVE."
    : GOTO 3000
4325  IF C$ = "GO PEDESTAL" AND CO(6) < > 0
        AND CO(6) < > 115 THEN PRINT "OKAY.
        "
    : GOTO 3000
4330  IF C$ = "GO PEDESTAL" THEN CO(6) = 11
        5
    : PRINT "THERE'S SOMETHING ON TOP OF TH
        E PEDESTAL"
    : GOTO 50000
```

```
4340  IF C$ = "GO HOLOGRAM" AND R = 4
        THEN PRINT "IT APPEARS TO BE SOME S
        ORT OF DETAILED  MAP, BUT THE CHARA
        CTERS AND LAYOUT ARE  INCOMPREHENSI
        BLE."
    : IF CO(3) = 1 THEN SPEED= 100
    : INVERSE
    : PRINT "THINK, EARTHLING."
    : NORMAL
    : SPEED= 255
4345  IF C$ = "GO HOLOGRAM" AND R = 4
        THEN 3000
4350  IF S2$ = " HOLOGRAM" AND R < > 4
        THEN PRINT A$(7)
    : GOTO 3000
4360  IF S2$ = " PANEL" AND (R = 2 OR R
        = 7 OR R = 24) THEN PRINT "OKAY."
    : GOTO 3000
4370  IF S2$ = " PANEL" THEN PRINT A$(7)
    : GOTO 3000
4380  IF S2$ = " AIRLOCK" AND R = 1 THEN PRINT
        "OKAY."
    : R = 2
    : GOTO 50000
4390  IF S2$ = " AIRLOCK" THEN PRINT A$(1)
    : GOTO 3000
4400  IF S2$ = " DOOR" THEN PRINT "OKAY."
    : GOTO 3000
4410  IF S2$ = " SHIP" AND R = 1 THEN R
        = 2
    : GOTO 50000
4420  IF S2$ = " SHIP" AND R = 2 AND D(10) =
        1 THEN R = 3
    : GOTO 50000
4430  IF S2$ = " SHIP" AND R = 2 THEN
        PRINT A$(5)
    : GOTO 3000
4440  IF S2$ = " SHIP" THEN PRINT A$(1)
    : GOTO 3000
4450  IF CO(3) = 107 AND LEFT$ (S2$,4) = "
        PED" AND R = 7 THEN PRINT "OKAY. TH
        AT IS WHERE THE CRYSTAL IS."
    : GOTO 3000
4460  IF LEFT$ (S2$,4) = " PED" AND R = 7 THEN
        PRINT "OKAY."
    : GOTO 3000
4470  IF LEFT$ (S2$,4) = " BED" THEN
        PRINT "SERIES R ROBOTS HAVE NO NEED
         OF SLEEP."
    : GOTO 3000
4480  IF LEFT$ (S2$,6) = " CHAIR" AND R
        = 15 THEN PRINT "OKAY."
    : PRINT A$(22)
    : GOTO 3000
4490  IF LEFT$ (S2$,6) = " CHAIR" THEN
        PRINT A$(7)
    : GOTO 3000
```

```
4500  IF S2$ = " TUBES" AND (R = 13 OR R =
        14) THEN PRINT A$(22)
    : GOTO 3000
4510  IF S2$ = " TUBES" THEN PRINT A$(7)
    : GOTO 3000
4520  IF R = 21 AND ( LEFT$ (S2$,6) = " TAB
        LE" OR S2$ = " SCULLERY" OR
        LEFT$ (S2$,5) = " OVEN" OR S2$ = "
        CLOSET") THEN PRINT A$(22)
    : GOTO 3000
4530  IF R = 22 AND (S2$ = " COURT" OR
        LEFT$ (S2$,5) = " ROPE" OR S2$ = "
        POOL") THEN PRINT "OKAY."
    : GOTO 3000
4540  IF S2$ = " COURT" OR LEFT$ (S2$,5) =
        " ROPE" OR S2$ = " POOL" THEN
        PRINT A$(7)
    : GOTO 3000
4550  IF C$ < > "GO DEVICE" THEN 4660
4555  IF R = 5 THEN PRINT "IT IS INACCESSIB
        LE."
    : GOTO 3000
4560  IF R < > 4 THEN PRINT A$(7)
    : GOTO 3000
4570  PRINT "IT SEEMS TO BE SOME SORT OF TR
        ANSMITTINGDEVICE. IT APPEARS TO BE
        OPERATIVE.      PLEASE STAND BY."
    : GOSUB 63000
4580  PRINT "IT IS SENDING A MESSAGE IN AN
        ELECTRONICCODE. THE MAIN MEMORY BAN
        KS OF THE CON- TROL COMPUTER SHOULD
         BE ABLE TO TRANS-  LATE IT. I AM S
        WITCHING CONTROL TO MAIN CIRCUITS.
        PLEASE STAND BY."
    : GOSUB 63000
4590  INVERSE
4600  SPEED= 100
4610  PRINT "TO THE PEOPLE OF THE SYSTEM OF
         THE NINE PLANETS: THIS IS A MESSAG
        E FROM THE      UNITED FEDERATION OF
         KRENN."
    : PRINT
4620  PRINT "THIS SHIP IS A MICRONUCLEAR EX
        PLOSIVE   DEVICE. IT IS AIMED DIREC
        TLY AT YOUR     STAR, AND WILL EXPLO
        DE WHEN IT REACHES  A 100,000,000 M
        ILE RADIUS. THIS WILL PUTIT ALMOST
        DIRECTLY NEXT TO YOUR HOME    PLANE
        T."
4630  PRINT
    : PRINT "THE REASON FOR THIS ACTION IS
        SIMPLE.   NOW THAT YOU HAVE DEVELOP
        ED FASTER THAN LIGHT TECHNOLOGY YOU
         ARE A THREAT TO THEGALACTIC DOMINA
        NCE OF THE KRENN. THERE- FORE YOU M
        UST BE DESTROYED."
```

```
4640  PRINT
    : PRINT "FAREWELL, PEOPLE OF THE SYSTEM
        OF THE   NINE PLANETS."
    : NORMAL
    : PRINT
4650  SPEED= 255
    : PRINT "END TRANSMISSION."
    : GOTO 50000
4660  PRINT A$(9)
    : GOTO 3000
```

Line 4160 is one of a sort that will pop up frequently. Our program knows what GO means, but it needs more information to act upon it. This line delivers that message.

Since the entire two-letter-word section is comprised of nothing but GO, no other two-letter words are acceptable, hence line 4170. And 4180 quantifies S2$ as we just discussed above. Lines 4190–4220 introduce us to the concept of synonyms. If there are two or more ways of saying something, use them all, since all we have to do is reference the variations back to the first usage. Here we do that with GO NORTH referenced back to N, GO SOUTH back to S, and so forth. You'll see a lot more of this as we go along.

Lines 4240–4330 refer to situations encountered only in room 15, which explains line 4230. The first few of these lines refer to the somewhat confusing multidirectional doors I introduced into this room. They may not have been such a good idea not only because my players reported back to me that they found them to be a pain in the neck, but also because they required special programming space to handle them. Beware of such game elements that cause more difficulties than they may be worth. In any case, the same programming structure we applied to the N, S, E, and W section applies to these. Lines 4320–4330 introduce special GO situations like those we'll meet up with in the remainder of this section. There are both a panel and a pedestal in room 15. Line 4320 describes the panel if the player has decided to go to it. Lines 4325 and 4330 describe the pedestal, both with and without the crystal star, CO(6) .

The important thing to remember is that your player *must* be able to move *everywhere* there is to move. If you look at the on-screen description of room 15, for instance, you will see that it specifically mentions both the panel and pedestal. Even if these are inconsequential room decorations (as is the panel), it is imperative that the program nonetheless recognize their existence. It would be somewhat second-rate for the player to input GO PANEL, only to get a reply I DON'T KNOW WHAT A PANEL IS. It's there, so it has to be accounted for. This means a lot of somewhat tedious programming, but it also means that your game universe is complete, with all its components "real" insofar as both the player and the program are concerned. Line 4400 is the supreme example of this. Since there are doorways of one sort or another in every room, whenever a player says GO DOOR the program replies with OKAY, as if the robot has been moved up to the door. Nothing happens, as you can see, but at least the player thinks something happens, and that's the important thing.

You'll notice that there's no cowcatcher between 4330 and the rest of the GOs—i.e., there's no catchall barrier in the program to handle the fact that

the condition of being in room 15 is no longer met. That's because all the rest of the conditions apply equally to room 15 and to the rest of the rooms.

Lines 4340–4350 handle the hologram in room 4. The IF CO(3) = 1 business is the clue to the use of the crystal pentagon. Note that if the CO(3) = 1 condition is not met, Applesoft will not continue processing along this line through the colons and will continue to 4345.

Let us now digress for a minute. We've been dealing with something here all along without defining it because the operation has been so obvious as not to bear further discussion. But it's something we can't ignore any longer because we're going to start stretching it to its limits. That something is called Boolean logic.

When we say something like:

4350 IF S2$ = " HOLOGRAM" AND R< > 4 THEN PRINT A$(7): GOTO 3000

the AND statement is a Boolean operator. The computer reads this statement and, provided both AND conditions are met, does what is asked of it regardless of what those conditions are. As far as the computer is concerned, we have spoken the truth, so the computer applies a TRUE to that situation (which, if you're really interested, is a value of 1; FALSE would be a value of 0) and does what we ask of it. If either one of those conditions is not met, the computer does not do what we ask of it.

OR is another Boolean operator. In line 4360 we say:

4360 IF S2$ = " PANEL" AND (R = 2 OR R = 7 OR R = 24) THEN PRINT "OKAY." : GOTO 3000

which incorporates both AND and OR. With OR all you have to do is get one TRUE for the computer to obey your commands. In line 4360 we compound our Booleania by first getting some truth out of what's in the parentheses and adding this truth to the truth of S2$. When the computer comes across this line it first looks inside the parentheses—parenthetical statements take precedence in the actual processing. For the computer to come up with a TRUE for the ORs all we need is one of these values of R to be accurate. If none of them is accurate, we get a FALSE. After handling the parentheses the computer goes back and measures the rest of the line. Now it's looking for truth on both sides of the AND. If S2$ does equal " PANEL" at this point, but R equals none of these values, nothing will happen because the FALSE overrides the TRUE in the AND situation. Two TRUEs equal truth. And as in the real world, one FALSE plus one TRUE do not equal the truth, and two FALSEs also do not equal the truth.

As I said, most of the time you handle OR and AND situations without the slightest concern for the Booleanism of the thing, but I mention it now because of the way it is compounded by the parentheses in 4360 (and also further along in both this and the poker program). Using this kind of seemingly straightforward logic can eventually lead to some pretty complicated-looking programming statements, so knowing what it's all about won't hurt. The interesting facet to all of this is that you can indeed use parentheses for nonmathematical statements. We're used to seeing things such as:

X = INT(C)/( (Y+2)*(Z−Y) )

but we can also say things such as:

IF ( (A$ = "WHEATIES" OR B$ = "SALAMI") AND (C$ = "MEASLES" OR D$ = "PLAGUE") ) AND E$ = "TIME IS ON OUR SIDE"

THEN PRINT "WHAT???"
and have the computer make sense of it.

The last thing worth noting here is that the computer measures all parts of Boolean statements, even after the condition we're looking for is met. For example, let's say we have this line:
IF X = 5 OR (10/Y > 0) THEN Z = 11
and let's further assume at this point X does equal 5 and Y equals 0. First off, the computer measures X = 5 and finds truth. The problem is, despite having satisfied the conditional—if X = 5 it doesn't matter anymore whether 10/Y>0 because of the OR—the computer nonetheless goes ahead and measures 10/Y>0 anyhow. And what would happen here is a break in the program for a "division by 0" error. Beware of such potential disasters in your logic evaluations.

So much for Boolean logic for now.

The rest of this section in "Space Derelict!" pretty much does what we've been doing all along, measuring for the correct situations and either acting upon them if the criteria are fulfilled or sending them back to the player if they're not. Therefore I won't go into these lines any further, with two exceptions. First, there are some beds scattered around in "Space Derelict!"; hence line 4470. This is, in fact, a joke, and the more jokes you can put into your game, the better. But this line also makes another point. You'll notice that here I have used the LEFT$ statement on BED. This is to allow the possibility that the player might have originally typed in GO BEDS as well as GO BED, given the game situation. This way, the program handles both possibilities. Lines 4450 and 4460 do a similar job on PED as in PEDESTAL. PEDESTAL is an easy word to mistype, and we can accept it on the basis of the first three letters since we know that there are no other words around with the same first three letters. It's nice to provide this little service to the player, whose sticky fingers may be burning such a hole in the computer as not always to get the right thing down, even though, God bless him, he was close. So we give him the benefit of the doubt, and good luck to him.

The other thing out of the ordinary is lines 4570–4650. This, as you can see, is the message from our friends the Krenn. Here we've combined some description from our puppet robot with the actual INVERSE, slow-speed Krenn transmission. Not bad, if I do say so myself. The GOSUB 63000 refers to a pause you should type in right now:

***"Space Derelict!"***

```
63000 FOR PAUSE = 1 TO 2500
    : NEXT
    : RETURN
```

Finally, line 4660 cowcatches everything that fell through the cracks up to here, printing A$(9) and sending us back to 3000, where it all began.

## THREE-LETTER WORDS

We're starting to run out of new tricks in our adventure programming bag, which means in a way that we are becoming old dogs. We will look at a few more little things that make the adventure design easier overall, but we've

now covered all the major concepts for the design of this kind of game—except, of course, for graphics, which you can make of what you will, and which we'll discuss briefly at the end of Chapter 6. Not that we've covered all there is to know about game programming—the poker program contains dozens of cans of all-new worms—and for that matter we're not even near the end of filling in "Space Derelict!" But we are over the hump as far as the technical concepts are concerned. Let's run the three-letter words up the flagpole and see who salutes.

**"Space Derelict!"**

```
4670  REM  THREE LETTER WORDS
4680  IF C$ = "RUN" THEN PRINT "I NEED A DI
        RECTION."
    : GOTO 3000
4690  IF LEFT$ (C$,3) = "RUN" THEN C$ = "GO
        " + MID$ (C$,4)
    : GOTO 3000
4700  IF C$ = "CRY" THEN PRINT "SERIES R RO
        BOTS ARE DESIGNED NOT TO CRY AS A P
        REVENTATIVE AGAINST RUST."
    : GOTO 3000
4710  IF LEFT$ (C$,3) = "GET" THEN C$ = "TA
        KE" + MID$ (C$,4)
    : GOTO 3080
4720  IF LEFT$ (C$,3) = "TIE" AND CO(12) =
        1 THEN PRINT "I DON'T SEE MUCH POIN
        T TO THAT."
    : GOTO 3000
4730  IF LEFT$ (C$,3) = "TIE" THEN
        PRINT A$(8)
    : GOTO 3000
4740  IF C$ = "SAY" THEN PRINT A$(1)
    : GOTO 3000
4750  IF C$ < > "SAY OPEN" THEN 5000
4760  PRINT "OKAY."
    : PRINT "OPEN"
4770  IF CO(3) < > 1 THEN PRINT "NOTHING HA
        PPENS."
    : GOTO 3000
4780  IF R = 6 AND D(1) = 0 THEN D(1) = 1
    : PRINT "THE NORTHERN DOOR HAS OPENED."

    : GOTO 3000
4790  IF R = 7 AND D(2) = 0 THEN D(2) = 1
    : PRINT "THE NORTH DOOR HAS OPENED."
    : GOTO 3000
4800  IF R = 11 THEN D(11) = 1
    : PRINT "I'M WILLING TO GO IN IF YOU TH
        INK IT'S  ADVISABLE."
    : GOTO 3000
4810  IF R = 10 AND D(3) = 1 AND D(4) = 1 THEN
        PRINT "NOTHING HAPPENS."
    : GOTO 3000
```

```
4820  IF R = 10 AND D(4) = 0 THEN D(4) = 1
    : PRINT "THE NORTH DOOR HAS OPENED."
4830  IF R = 10 AND D(3) = 0 THEN D(3) = 1
    : PRINT "THE SOUTH DOOR HAS OPENED."
4840  IF R = 10 THEN 3000
4850  IF R = 22 AND D(4) = 0 THEN D(4) = 1
    : PRINT "THE SOUTH DOOR HAS OPENED."
    : GOTO 3000
4860  IF R = 16 AND D(16) = 0 THEN D(16) =
        1
    : PRINT "THE EAST DOOR HAS OPENED."
    : GOTO 3000
4870  IF R = 17 THEN PRINT "ALL THE DOORS A
        RE OPEN."
    : FOR X = 14 TO 17
    : D(X) = 1
    : NEXT X
    : GOTO 3000
4880  IF R = 13 AND D(15) = 0 THEN D(15) =
        1
    : PRINT "THE DOOR IS NOW OPEN."
    : GOTO 3000
4890  IF R = 14 AND D(14) = 0 THEN D(14) =
        1
    : PRINT "THE DOOR IS NOW OPEN."
    : GOTO 3000
4900  IF R < > 15 THEN 4960
4910  IF D(16) = 0 THEN D(16) = 1
    : PRINT "THE WEST(N) DOOR HAS OPENED."
4920  IF D(15) = 0 THEN D(15) = 1
    : PRINT "THE WEST(S) DOOR HAS OPENED."
4930  IF D(17) = 0 THEN D(17) = 1
    : PRINT "THE EAST(N) DOOR HAS OPENED."
4940  IF D(14) = 0 THEN D(14) = 1
    : PRINT "THE EAST(S) DOOR HAS OPENED."
4950  PRINT "OKAY."
    : GOTO 3000
4960  IF R = 17 AND D(17) = 0 THEN D(17) =
        1
    : PRINT "THE WEST DOOR IS NOW OPEN."
    : GOTO 3000
4970  IF R = 18 AND D(6) = 0 THEN D(6) = 1
    : PRINT "THE SOUTH DOOR IS NOW OPEN."
    : GOTO 3000
4980  IF R = 19 AND D(6) = 0 THEN D(6) = 1
    : PRINT "THE DOOR IS NOW OPEN."
    : GOTO 3000
5000  IF LEFT$ (C$,3) = "SAY" THEN
        PRINT "OKAY."
    : PRINT MID$ (C$,4)
    : PRINT "NOTHING HAPPENS."
    : GOTO 3000
5010  IF R = 22 AND CO(12) = 0 AND CO(9) =
        1 AND C$ = "CUT ROPE" THEN
        PRINT "OKAY."
    : CO(12) = 122
    : GOTO 50000
```

```
5020  IF C$ = "CUT ROPE" AND CO(9) < > 1 THEN
        PRINT A$(6)
    : GOTO 3000
5030  IF C$ = "CUT ROPE" AND (CO(12) <
        > 100 + R OR (R < > 22 AND CO(12) =
        0) OR CO(12) < > 1) THEN PRINT A$(1
        )
    : GOTO 3000
5040  IF C$ = "CUT" THEN PRINT A$(1)
    : GOTO 3000
5050  IF LEFT$ (C$,3) = "CUT" THEN
        PRINT "I CANNOT CUT THAT."
    : GOTO 3000
5060  PRINT A$(2) + LEFT$ (C$,3)
    : GOTO 3000
```

This time we'll concentrate just on special cases.

In "Space Derelict!" there is no advantage to the player in going somewhere quickly, so I've relegated the word RUN to synonym stature in line 4690. Since we've already excluded RUN as a complete command in 4680, we know that if RUN is the first word, then the length of C$ is greater than 3. So line 4690 transmutes the RUN command into a GO command by redefining C$ as GO plus all the rest of the original C$ starting with the space. Then the program branches to 4150, which is the beginning of the two-letter first-word commands, since that exactly defines our re-created C$. The more synonyms you have, the more user-friendly your game, and as you can see there's not much work in it for you as the designer.

Line 4700 is another joke. I point this out because it might look like it, but a joke it is. If your jokes are any better, let me know.

Line 4710 is another synonym, and I only mention it because I have played games where TAKE was not an acceptable command, which amazes me as it took me a long time to come up with GET as a substitute. But apparently it is not unusual for some adventurers to operate the other way around entirely, and it never occurs to them to TAKE anything. In any case, both commands are essential in your program.

Line 4720 introduces the "intelligent" element of our puppet robot once again. Since in our game concept the robot possesses some sort of native intelligence, it makes sense that he perform some actions on his own (as with turning on the flashlight in room 23) and also that he occasionally has a chance to get his two cents in. This concept is fairly nascent in "Space Derelict!" and is one you can develop quite a bit further in an original game to make quite an interesting variation on your basic adventure. Line 4730, of course, works from the logical exclusions of 4720.

The use of the command SAY in your adventure is entirely optional, but magic words and the like come up often enough to have made it worth including an arrangement of this sort in "Space Derelict!" Since saying the word OPEN is the clincher in this game, you can see how this command has its effect in lines 4750–4980. Line 5000 acts both as a cowcatcher from the SAY OPEN routine and a catchall for any game where SAYing something is going to have no effect. Line 5000 comes back with an OKAY and then prints C$ starting with the space so that the player sees repeated on the screen what he wanted to say. So if the player commands:

SAY ABRACADAVER
the computer replies with
OKAY
ABRACADAVER
NOTHING HAPPENS
and back we go to the beginning of 3000. When we say here that nothing happens, we really mean it, but it looks good, and it may fool the player into saying everything that comes into his head, which will be one more step each time on the road away from solving your puzzle, which is something we want to see as often as possible.

Within the SAY OPEN section there is an interesting little phenomenon in lines 4820–4840. If the player is in room 10 and says the magic word, there are two doors, D(4) and D(3), that may or may not respond to him. If D(4) is closed, the command will open that door, and the player will be so notified in line 4820. Then the processing continues to 4830. If D(3) is closed, then this door too will open and again the player will be so notified. Line 4840 sends him back to 3000. As you can see, there are four possibilities here, depending on the values of D(4) and D(3). The player will see on the screen that one has opened, the other has opened, both have opened, or neither have opened, depending on the situation. This kind of programming structure allows for multiple possibilities where such are desirable, and the nice part of it is, they appear to occur virtually simultaneously on the video monitor.

Line 5030 contains a nasty bit of Boolean operation of parentheses within parentheses that really makes for an interesting situation. Broken down into its component parts, it looks like this:
IF AND ( OR (AND) OR ) THEN
Once again we're looking for truth. First the computer will determine the truth of the innermost AND. Those mean-looking ORs allow that if any one of the statements within these next parentheses is true, then the whole parenthetical statement becomes true. So if both of the halves of the innermost AND are true, or if either of the other OR possibilities within the parentheses is true, than the whole parenthetical statement gets ranked as truth. Now all that has to be determined is if both halves of the outermost AND contain truth. If so, then processing will proceed along this line. What we're trying to do in this line could have been accomplished in three lines, but the use of Boolean logic allows us to squeeze it all into one. I've been relatively spare with this sort of programming in "Space Derelict!" because it obviously can become very sticky very easily. It's not the easiest thing in the world to explain in the abstract, and more to the point, it makes it harder to follow what's going on in the program. But the more of this kind of programming you can do yourself, the more efficient your program will be, leaving more space for nastiness in your complications rather than complications in your programs.

## A FEW CHOICE FOUR-LETTER WORDS

No-man's-land continues now, with line after line of programming drudgery. But each line serves a purpose, and once you have a list of all the words you want to use in your adventure, you'll be surprised how quickly this phase of the programming can actually go. With one exception, all of the four-letter words below are rated G.

```
5070  REM  4 LETTER WORDS
5080  S$ = LEFT$ (C$,4)
    : REM  TO SAVE TYPING TIME
5090  IF LEN (C$) > 4 THEN S2$ = MID$ (C$,5
      )
5100  IF S$ = "READ" THEN PRINT "THERE IS N
      OTHING READABLE HERE."
    : GOTO 3000
5110  IF S$ = "PUSH" THEN C$ = "PRESS" + S2
      $
    : GOTO 7210
5120  IF S$ < > "TAKE" THEN 5530
5130  IF C$ = "TAKE" THEN PRINT A$(9)
    : GOTO 3000
5140  IF LEFT$ (C$,8) = "TAKE INV" THEN 131
      00
5150  IF S$ = "TAKE" AND W = 5 THEN
      PRINT "I CAN'T DO THAT. I AM CARRYI
      NG TOO MUCH.DO YOU WISH AN INVENTOR
      Y?"
    : GOTO 3000
5160  GOTO 5180
5170  W = W + 1
    : GOTO 50000
5180  IF S2$ = " METAL" AND CO(1) = 100
      + R THEN CO(1) = 1
    : GOTO 5170
5190  IF C$ = "TAKE METAL" THEN PRINT A$(7)

    : GOTO 3000
5200  IF S2$ = " PENTAGON" AND CO(2) = 100 +
      R THEN CO(2) = 1
    : GOTO 5170
5210  IF C$ = "TAKE PENTAGON" THEN
      PRINT A$(7)
    : GOTO 3000
5220  IF S2$ = " CRYSTAL" AND CO(3) = 100 +
      R THEN CO(3) = 1
    : GOTO 5170
5230  IF C$ = "TAKE CRYSTAL" THEN PRINT A$(
      7)
    : GOTO 3000
5240  IF S2$ = " WEAPON" AND CO(4) = 100 +
      R THEN CO(4) = 1
    : GOTO 5170
5250  IF C$ = "TAKE WEAPON" THEN PRINT A$(7
      )
    : GOTO 3000
5260  IF S2$ = " BOX" AND CO(5) = 100 + R THEN
      CO(5) = 1
    : GOTO 5170
5270  IF C$ = "TAKE BOX" THEN PRINT A$(7)
    : GOTO 3000
```

```
5280  IF S2$ = " STAR" AND CO(6) = 115
        AND CO(1) = 1 AND R = 15 THEN CO(6)
        = 1
    : PRINT "OKAY."
    : GOTO 5170
5290  IF S2$ = " STAR" AND CO(6) = 100 + R AND
        R < > 15 THEN CO(6) = 1
    : PRINT "OKAY."
    : GOTO 5170
5300  IF S2$ = " STAR" AND CO(6) = 100 + R THEN
        PRINT "IT'S ATTACHED SOMEHOW. I NEE
        D SOMETHING TO LOOSEN IT."
    : GOTO 3000
5310  IF S2$ = " STAR" THEN PRINT A$(7)
    : GOTO 3000
5330  IF S2$ = " LASER" AND CO(7) = 100
        + R THEN CO(7) = 1
    : GOTO 5170
5340  IF C$ = "TAKE LASER" THEN PRINT A$(7)

    : GOTO 3000
5350  IF LEFT$ (S2$,5) = " FLAS" AND CO(8) =
        100 + R THEN CO(8) = 1
    : GOTO 5170
5360  IF LEFT$ (C$,11) = "TAKE FLASHL"
        THEN PRINT A$(7)
    : GOTO 3000
5370  IF S2$ = " KNIFE" AND CO(9) = 100
        + R THEN CO(9) = 1
    : GOTO 5170
5380  IF C$ = "TAKE KNIFE" THEN PRINT A$(7)

    : GOTO 3000
5390  IF S2$ = " BUCKET" AND CO(10) = 100 +
        R THEN CO(10) = 1
    : GOTO 5170
5400  IF S2$ = " BUCKET" AND CO(15) = 100 +
        R THEN CO(15) = 1
    : GOTO 5170
5410  IF C$ = "TAKE BUCKET" THEN PRINT A$(7
        )
    : GOTO 3000
5420  IF S2$ = " ROBOT" AND CO(11) = 100 +
        R THEN CO(11) = 1
    : GOTO 5170
5430  IF S2$ = " ROBOT" AND CO(13) = 100 +
        R THEN PRINT A$(1)
    : GOTO 3000
5440  IF S2$ = " ROBOT" AND CO(16) = 100 +
        R THEN CO(16) = 1
    : GOTO 5170
5450  IF C$ = "TAKE ROBOT" THEN PRINT A$(1)

    : GOTO 3000
5460  IF S2$ = " ROPE" AND CO(12) = 100
        + R THEN CO(12) = 1
    : GOTO 5170
```

```
5470  IF S2$ = " ROPE" AND CO(12) = 0
        AND R = 22 THEN PRINT A$(8)
    : GOTO 3000
5480  IF C$ = "TAKE ROPE" THEN PRINT A$(7)
    : GOTO 3000
5490  IF C$ = "TAKE WATER" AND R = 22
        AND CO(14) = 122 AND CO(10) = 1
        THEN CO(10) = 0
    : CO(15) = 1
    : PRINT "OKAY."
    : GOTO 50000
5500  IF C$ = "TAKE WATER" AND R = 22
        AND CO(14) = 122 THEN PRINT A$(1)
    : GOTO 3000
5510  IF C$ = "TAKE WATER" THEN PRINT A$(6)

    : GOTO 3000
5520  PRINT "I CANNOT " + C$
    : GOTO 3000
5530  IF C$ = "SAVE" OR C$ = "SAVE GAME" THEN
        30000
5540  IF S$ < > "QUIT" THEN 5590
5550  INPUT "DO YOU WISH TO TRY AGAIN?";C$
5560  IF C$ = "Y" OR C$ = "YES" THEN
        CLEAR
    : GOTO 30
5570  IF C$ = "N" OR C$ = "NO" THEN END
5580  GOTO 5550
5590  IF C$ = "FUCK YOU" THEN PRINT "PLEASE
        MAINTAIN A PROPER ATTITUDE. THE  F
        ATE OF THE EARTH RESTS IN YOUR HAND
        S."
    : GOTO 3000
5600  IF S$ = "KICK" THEN PRINT A$(1)
    : GOTO 3000
5610  IF S$ = "HELP" THEN PRINT "PERHAPS WE
        ARE NOT MANIPULATING THE     ITEMS
        AT HAND CORRECTLY."
    : GOTO 3000
5630  IF C$ = "KILL ROBOT" AND R = 23
        THEN PRINT A$(8)
    : GOTO 3000
5640  IF S$ = "KILL" THEN PRINT A$(1)
    : GOTO 3000
5650  IF S$ < > "OPEN" THEN 6180
5660  IF C$ = "OPEN BOX" AND (CO(5) = 1
        OR CO(5) = 100 + R) AND CO(6) = 5 THEN
        PRINT "THE CRYSTAL STAR IS IN IT."
    : GOTO 3000
5670  IF C$ = "OPEN BOX" AND (CO(5) = 1
        OR CO(5) = 100 + R) THEN PRINT "OKA
        Y."
    : GOTO 3000
5680  IF C$ = "OPEN BOX" THEN PRINT A$(7)
    : GOTO 3000
5690  IF C$ < > "OPEN DOOR" THEN PRINT A$(1)
    : GOTO 3000
```

```
5700  IF R = 1 THEN PRINT "THERE IS NO DOOR
        HERE."
    : GOTO 3000
5710  IF R < 4 THEN PRINT A$(8)
    : GOTO 3000
5720  IF R < 6 THEN PRINT "THERE IS NO DOOR
        HERE TO OPEN."
    : GOTO 3000
5730  IF R = 6 AND D(1) = 1 AND D(2) = 1 THEN
      PRINT "THE DOORS HERE ARE OPEN."
    : GOTO 3000
5740  IF R = 6 THEN PRINT A$(8)
    : GOTO 3000
5750  IF R = 7 AND D(2) = 0 THEN PRINT A$(8
      )
    : GOTO 3000
5760  IF R = 7 THEN PRINT "THE DOOR IS ALRE
      ADY OPEN."
    : GOTO 3000
5770  IF R < 10 THEN PRINT "THERE IS NO DOO
      R HERE TO OPEN."
    : GOTO 3000
5780  IF R = 10 AND D(4) = 0 AND D(3) = 0 AND
      D(13) = 0 THEN PRINT A$(8)
    : GOTO 3000
5790  IF R = 10 AND D(4) = 1 THEN PRINT "TH
      E NORTH DOOR IS OPEN."
5800  IF R = 10 AND D(3) = 1 THEN PRINT "TH
      E SOUTH DOOR IS OPEN."
5810  IF R = 10 THEN 3000
5820  IF R = 11 THEN PRINT A$(1)
    : GOTO 3000
5830  IF R = 12 AND D(3) = 0 THEN PRINT A$(
      8)
    : GOTO 3000
5840  IF R = 12 THEN PRINT "THE DOOR IS OPE
      N."
    : GOTO 3000
5850  IF R = 13 AND D(15) = 0 THEN
      PRINT A$(8)
    : GOTO 3000
5860  IF R = 13 THEN PRINT "THE DOOR IS OPE
      N."
    : GOTO 3000
5870  IF R = 14 AND D(14) = 0 THEN
      PRINT A$(8)
    : GOTO 3000
5880  IF R = 14 THEN PRINT "THE DOOR IS OPE
      N."
    : GOTO 3000
5890  IF R < > 15 THEN 5940
5900  FOR X = 14 TO 17
5910  IF D(X) = 0 THEN PRINT A$(8)
    : X = 17
    : NEXT
    : GOTO 3000
5920  NEXT X
```

```
5930  PRINT "ALL THE DOORS HERE ARE OPEN."
    : GOTO 3000
5940  IF R = 16 AND D(16) = 0 THEN
        PRINT A$(8)
    : GOTO 3000
5950  IF R = 16 THEN PRINT "THE DOOR IS OPE
        N."
    : GOTO 3000
5960  IF R = 17 AND D(17) = 0 THEN
        PRINT A$(8)
    : GOTO 3000
5970  IF R = 17 THEN PRINT "THE DOOR IS OPE
        N."
    : GOTO 3000
5980  IF R = 18 AND D(5) = 0 AND D(6) = 0 THEN
        PRINT A$(8)
    : GOTO 3000
5990  IF R = 18 AND D(5) = 1 AND D(6) = 1 THEN
        PRINT "THE DOORS HERE ARE BOTH OPEN
        ."
    : GOTO 3000
6000  IF R = 18 AND D(5) = 1 AND D(6) = 0 THEN
        PRINT "THE SOUTH DOOR IS ALREADY OP
        EN. HOW      SHOULD I OPEN THE NORTH
         DOOR?"
    : GOTO 3000
6010  IF R = 18 THEN PRINT "THE NORTH DOOR
        IS OPEN. HOW SHOULD I    OPEN THE S
        OUTH ONE?"
    : GOTO 3000
6020  IF R = 19 AND D(6) = 0 THEN PRINT A$(
        8)
    : GOTO 3000
6030  IF R = 19 THEN PRINT "THE DOOR IS ALR
        EADY OPEN."
    : GOTO 3000
6040  IF R = 20 THEN PRINT "THERE ARE NO DO
        ORS HERE."
    : GOTO 3000
6050  IF R = 21 AND D(7) = 0 AND D(5) = 0 THEN
        PRINT A$(8)
    : GOTO 3000
6060  IF R = 21 AND D(7) = 1 AND D(5) = 1 THEN
        PRINT "THE DOORS HERE ARE ALREADY O
        PEN."
    : GOTO 3000
6070  IF R = 21 AND D(7) = 1 THEN PRINT "TH
        E NORTH DOOR IS OPEN. HOW DO I OPEN
           THE SOUTH ONE?"
    : GOTO 3000
6080  IF R = 21 THEN PRINT "THE SOUTH DOOR
        IS OPEN. HOW DO I OPEN   THE NORTH
        ONE."
    : GOTO 3000
6090  IF R = 22 AND D(8) = 0 AND D(4) = 0 THEN
        PRINT A$(8)
    : GOTO 3000
```

```
6100  IF R = 22 AND D(8) = 1 AND D(4) = 1 THEN
        PRINT "THE DOORS ARE ALREADY OPEN."

    : GOTO 3000
6110  IF R = 22 AND D(8) = 1 THEN PRINT "TH
        E NORTH DOOR IS OPEN. HOW DO I OPEN
           THE SOUTH DOOR?"
    : GOTO 3000
6120  IF R = 22 THEN PRINT "THE SOUTH DOOR
        IS OPEN. HOW DO I OPEN   THE NORTH
        DOOR?"
    : GOTO 3000
6140  IF R = 23 AND D(8) + D(9) + D(7) = 3 THEN
        PRINT "ALL THE DOORS IN THIS ROOM A
        RE OPEN."
    : GOTO 3000
6150  IF R = 23 THEN PRINT A$(8)
    : GOTO 3000
6160  IF R = 24 THEN PRINT "THE DOOR IS ALR
        EADY OPEN."
    : GOTO 3000
6180  IF S$ < > "LOOK" THEN 6920
6190  IF C$ = "LOOK" THEN PRINT A$(9)
    : GOTO 3000
6210  IF S2$ = " AIRLOCK" AND (R = 1 OR R =
        2) THEN PRINT A$(22)
    : GOTO 3000
6220  IF S2$ = " AIRLOCK" THEN PRINT A$(7)
    : GOTO 3000
6230  IF R = 1 AND S2$ = " ENTRANCE"
        THEN PRINT A$(22)
    : GOTO 3000
6240  IF S2$ = " ENTRANCE" THEN PRINT A$(7)

    : GOTO 3000
6250  IF S2$ = " SHIP" AND R = 1 THEN
        PRINT "IT IS ENORMOUS AND FIVE=SIDE
        D."
    : GOTO 3000
6260  IF S2$ = " SHIP" THEN PRINT A$(7)
    : GOTO 3000
6270  IF S2$ = " PANEL" AND (R = 2 OR R
        = 7 OR R = 15) THEN PRINT "IT IS SI
        X INCHES ACROSS. THE BUTTONS ARENOT
         LABELED."
    : GOTO 3000
6275  IF S2$ = " PANEL" AND R = 24 THEN PRINT
        "I CAN ADD NOTHING TO THE DESCRIPTI
        ON    ABOVE."
    : GOTO 3000
6280  IF S2$ = " PANEL" THEN PRINT A$(7)
    : GOTO 3000
6290  IF S2$ = " METAL" AND (CO(1) = 1
        OR CO(1) = 100 + R) THEN PRINT A$(1
        0)
    : GOTO 3000
6300  IF S2$ = " METAL" THEN PRINT A$(7)
    : GOTO 3000
```

```
6310  IF (S2$ = " ROBOT" OR S2$ = " OBJECT"
        ) AND R = 11 THEN PRINT "I HAVE NOT
        HING TO ADD TO THE ABOVE."
    : GOTO 3000
6315  IF R = 24 AND S2$ = " SLOT" THEN
        PRINT "IT IS ABOUT 7 INCHES LONG."
    : GOTO 3000
6316  IF S2$ = " SLOT" THEN PRINT A$(7)
    : GOTO 3000
6320  IF S2$ = " OBJECT" AND R < > 11
        THEN PRINT A$(9)
    : GOTO 3000
6330  IF S2$ = " PENTAGON" AND (CO(2) = 1 OR
        CO(2) = 100 + R) THEN PRINT A$(11)
    : GOTO 3000
6340  IF S2$ = " PENTAGON" THEN PRINT A$(7)

    : GOTO 3000
6350  IF S2$ = " CRYSTAL" AND (CO(3) = 1 OR
        CO(3) = 100 + R) THEN PRINT A$(12)
    : GOTO 3000
6360  IF S2$ = " CRYSTAL" THEN PRINT A$(7)
    : GOTO 3000
6370  IF C$ = "LOOK AROUND" OR C$ = "LOOK R
        OOM" THEN PRINT "I HAVE NOTHING TO
        ADD TO THE ABOVE."
    : GOTO 3000
6380  IF S2$ = " WEAPON" AND (CO(4) = 1
        OR CO(4) = 100 + R) THEN PRINT A$(1
        3)
    : GOTO 3000
6390  IF S2$ = " WEAPON" THEN PRINT A$(7)
    : GOTO 3000
6400  IF S2$ = " HOLOGRAM" OR S2$ = " DEVIC
        E" THEN PRINT "I COULD TELL YOU MOR
        E IF I WERE CLOSER  TO IT."
    : GOTO 3000
6410  IF S2$ = " PEDESTAL" AND R = 7
        THEN PRINT "IT IS ABOUT FOUR FEET H
        IGH. THE CRYSTAL IS SEATED ON TOP O
        F IT."
    : GOTO 3000
6420  IF S2$ = " PEDESTAL" AND R = 5 AND CO
        (5) = 105 THEN PRINT "IT IS ABOUT F
        OUR FEET HIGH. THAT IS     WHERE TH
        E BOX IS."
    : GOTO 3000
6430  IF S2$ = " PEDESTAL" AND R = 5
        THEN PRINT "IT IS ABOUT FOUR FEET H
        IGH."
    : GOTO 3000
6440  IF S2$ = " PEDESTAL" AND R = 15
        AND CO(6) = 115 THEN PRINT "IT IS F
        OUR FEET HIGH. THAT IS WHERE THE ST
        AR IS."
    : GOTO 3000
```

```
6450  IF S2$ = " PEDESTAL" AND R = 15
        THEN PRINT "IT IS FOUR FEET HIGH."
    : GOTO 3000
6460  IF S2$ = " PEDESTAL" THEN PRINT A$(7)

    : GOTO 3000
6465  IF S2$ = " WALL" THEN PRINT A$(22)
6470  IF S2$ = " BOX" AND (CO(5) = 1 OR CO(
        5) = 100 + R) THEN PRINT A$(14)
6480  IF S2$ = " BOX" AND (CO(5) = 1 OR CO(
        5) = 100 + R) AND CO(6) = 5 THEN PRINT
        "THE CRYSTAL STAR IS IN IT."
6490  IF S2$ = " BOX" AND (CO(5) = 1 OR CO(
        5) = 100 + R) THEN 3000
6500  IF S2$ = " BOX" THEN PRINT A$(7)
    : GOTO 3000
6510  IF S2$ = " CEILING" AND R = 5 THEN PRINT
        "IT'S TOO HIGH UP FOR ME TO DESCRIB
        E IT  IN ANY GREATER DETAIL."
    : GOTO 3000
6520  IF S2$ = " CEILING" THEN PRINT A$(22)

    : GOTO 3000
6530  IF S2$ = " DEVICE" AND R = 5 THEN S2$
        = " CEILING"
    : GOTO 6510
6540  IF S2$ = " PASSAGEWAY" THEN PRINT A$(
        22)
    : GOTO 3000
6550  IF S2$ = " FOOTPRINTS" AND (R = 8
        OR R = 9 OR R = 10) THEN PRINT A$(2
        2)
    : GOTO 3000
6560  IF S2$ = " FOOTPRINTS" THEN PRINT A$(
        1)
    : GOTO 3000
6570  IF S2$ = " DUST" AND R = 8 THEN
        PRINT A$(22)
    : GOTO 3000
6580  IF S2$ = " DUST" THEN PRINT A$(1)
    : GOTO 3000
6590  IF LEFT$ (S2$,7) = " CROSSR" AND R =
        9 THEN PRINT A$(22)
    : GOTO 3000
6600  IF LEFT$ (S2$,7) = " CROSSR" THEN PRINT
        A$(1)
    : GOTO 3000
6610  IF S2$ = " STAR" AND (CO(6) = 1 OR CO
        (6) = 100 + R) THEN PRINT A$(15)
    : GOTO 3000
6620  IF S2$ = " STAR" THEN PRINT A$(7)
    : GOTO 3000
6630  IF S2$ = " DOOR" THEN PRINT A$(22)
    : GOTO 3000
```

```
6640  IF S2$ = " BED" AND (R = 12 OR R = 19)
      THEN PRINT "I'M NOT SURE IF IT
      REALLY IS A BED, BUT THAT'S MY BEST
       GUESS."
    : GOTO 3000
6650  IF S2$ = " BED" THEN PRINT A$(7)
    : GOTO 3000
6660  IF LEFT$ (S2$,6) = " CHAIR" AND R
        = 15 THEN PRINT A$(22)
    : GOTO 3000
6670  IF LEFT$ (S2$,6) = " CHAIR" THEN
        PRINT A$(7)
    : GOTO 3000
6680  IF ( LEFT$ (S2$,7) = " COMPUT" OR LEFT$
        (S2$,7) = " CONSOL") AND R = 15
        THEN PRINT A$(22)
    : GOTO 3000
6690  IF LEFT$ (S2$,7) = " COMPUT" OR
        LEFT$ (S2$,7) = " CONSOL" THEN
        PRINT A$(7)
    : GOTO 3000
6700  IF S2$ = " BUTTONS" AND (R = 15 OR R =
        2 OR R = 7) THEN PRINT A$(22)
    : GOTO 3000
6710  IF S2$ = " BUTTONS" THEN PRINT A$(7)
    : GOTO 3000
6720  IF S2$ = " LASER" AND (CO(7) = 1
        OR CO(7) = 100 + R) THEN PRINT A$(2
        3)
    : GOTO 3000
6730  IF S2$ = " LASER" THEN PRINT A$(7)
    : GOTO 3000
6740  IF S2$ = " FLASHLIGHT" AND (CO(8)
        = 1 OR CO(8) = 100 + R) THEN
        PRINT A$(16)
    : GOTO 3000
6750  IF S2$ = " FLASHLIGHT" THEN PRINT A$(
        7)
    : GOTO 3000
6760  IF S2$ = " KNIFE" AND (CO(9) = 121 OR
        CO(9) = 1) THEN PRINT A$(17)
    : GOTO 3000
6770  IF S2$ = " KNIFE" THEN PRINT A$(7)
    : GOTO 3000
6780  IF R = 22 AND (S2$ = " COURT" OR
        LEFT$ (S2$,5) = " ROPE") THEN
        PRINT A$(22)
    : GOTO 3000
6790  IF S2$ = " COURT" OR LEFT$ (S2$,5) =
        " ROPE" THEN PRINT A$(7)
    : GOTO 3000
6800  IF S2$ = " POOL" AND R = 22 THEN
        PRINT "IT IS VERY COLD ON THIS SHIP
        . APPARENTLYTHE WATER IN IT HAS COM
        PLETELY FROZEN.  IT IS FILLED WITH
        ICE."
    : GOTO 3000
```

```
6810  IF S2$ = " POOL" THEN PRINT A$(7)
    : GOTO 3000
6820  IF S2$ = " BUCKET" AND (CO(10) = 1 OR
        CO(10) = 100 + R) THEN PRINT A$(18)

    : GOTO 3000
6830  IF S2$ = " BUCKET" AND (CO(15) = 1 OR
        CO(15) = 100 + R) THEN PRINT A$(18)

    : GOTO 3000
6840  IF S2$ = " BUCKET" THEN PRINT A$(7)
    : GOTO 3000
6850  IF S2$ = " ROBOT" AND (CO(11) = 1
        OR CO(11) = 100 + R) THEN PRINT A$(
        19)
    : GOTO 3000
6860  IF S2$ = " ROBOT" AND (CO(13) = 1
        OR CO(13) = 100 + R) THEN PRINT A$(
        20)
    : GOTO 3000
6870  IF S2$ = " ROBOT" AND (CO(16) = 1
        OR CO(16) = 100 + R) THEN PRINT "IT
         IS DEFINITELY OUT OF COMMISSION. T
        HE WATER WAS EFFECTIVE."
    : GOTO 3000
6880  IF S2$ = " ROBOT" THEN PRINT A$(7)
    : GOTO 3000
6890  IF S2$ = " ROPE" OR S2$ = " WATER" THEN
        PRINT S2$ + " IS" + S2$
    : GOTO 3000
6900  PRINT "I AM NOT CERTAIN WHAT YOU MEAN
         BY THE   TERM" + S2$
    : GOTO 3000
6920  IF S$ < > "DROP" THEN 7130
6930  GOTO 6950
6940  W = W - 1
    : GOTO 50000
    : REM  ALL SUCCESSFUL DROPS END HERE AN
        D 50000 WILL ADJUST R$
6950  IF S2$ = " METAL" AND CO(1) = 1
        THEN CO(1) = 100 + R
    : GOTO 6940
6960  IF S2$ = " PENTAGON" AND CO(2) = 1 THEN
        CO(2) = 100 + R
    : GOTO 6940
6970  IF S2$ = " CRYSTAL" AND CO(3) = 1
        THEN CO(3) = 100 + R
    : GOTO 6940
6980  IF S2$ = " WEAPON" AND CO(4) = 1
        THEN CO(4) = 100 + R
    : GOTO 6940
6990  IF S2$ = " BOX" AND CO(5) = 1 THEN CO
        (5) = 100 + R
    : GOTO 6940
7000  IF S2$ = " STAR" AND CO(6) = 1
        THEN CO(6) = 100 + R
    : GOTO 6940
```

```
7010  IF S2$ = " LASER" AND CO(7) = 1
        THEN CO(7) = 100 + R
    : GOTO 6940
7020  IF S2$ = " FLASHLIGHT" AND CO(8) = 1 THEN
        CO(8) = 100 + R
    : GOTO 6940
7030  IF S2$ = " KNIFE" AND CO(9) = 1
        THEN CO(9) = 100 + R
    : GOTO 6940
7040  IF S2$ = " BUCKET" AND CO(10) = 1
        THEN CO(10) = 100 + R
    : GOTO 6940
7050  IF S2$ = " BUCKET" AND CO(15) = 1
        THEN CO(15) = 100 + R
    : GOTO 6940
7060  IF S2$ = " ROBOT" AND CO(11) = 1
        THEN CO(11) = 100 + R
    : GOTO 6940
7070  IF S2$ = " ROPE" AND CO(12) = 1
        THEN CO(12) = 100 + R
    : GOTO 6940
7080  IF S2$ = " ROBOT" AND CO(13) = 1
        THEN CO(13) = 100 + R
    : GOTO 6940
7090  IF S2$ = " WATER" AND CO(15) = 1
        THEN CO(10) = 1
    : PRINT "IT FREEZES IMMEDIATELY."
    : CO(15) = 0
    : GOTO 50000
7110  IF S2$ = " ROBOT" AND CO(16) = 1
        THEN CO(16) = 100 + R
    : GOTO 6940
7120  PRINT A$(1)
    : GOTO 3000
7130  IF S$ = "FIND" THEN PRINT A$(8)
    : GOTO 3000
7140  IF S$ < > "MELT" THEN 7190
7150  IF C$ = "MELT ICE" AND CO(7) = 1
        AND R = 22 THEN CO(14) = 100 + R
    : GOTO 50000
7160  IF S2$ < > " ICE" THEN PRINT "I CAN'T
         MELT " + S2$
    : GOTO 3000
7170  IF CO(7) < > 1 THEN PRINT A$(6)
    : GOTO 3000
7180  IF R < > 22 THEN PRINT "I DON'T SEE H
        OW I CAN DO THAT SUCCESS-  FULLY AT
         THIS TIME."
    : GOTO 3000
7190  IF C$ = "LOCK DOOR" THEN PRINT "THERE
         DOESN'T SEEM TO BE ANY WAY TO DO
        THAT. APPARENTLY ONCE THEY'RE OPEN
        THEY STAY OPEN."
    : GOTO 3000
7200  IF S$ = "LOCK" THEN PRINT A$(1)
    : GOTO 3000
7205  PRINT A$(2) + S$
    : GOTO 3000
```

Right off the top we're going to add a new wrinkle that comes in handy when we have a lot of words in a particular length grouping, as we do with the four-letter ones. Line 5080 now gives us a variable we can use that will be the four-letter first word, just as S2$ (which is stylistically upgraded in 5090) gives us everything after the first word. If you're a curious sort and had been wondering where the 2 had come from in S2$, now you know.

For most of this section once again I won't go into every nook and cranny of every line, but I will point out the highlights. Line 5100, for instance, covers all C$ lengths; it doesn't matter what S2$ our player wants to read, because there is nothing readable on this spaceship, and that is that. The fact that the hologram is unreadable is readily available to the player, and that's the only item even remotely decipherable.

For the record, line 5140 is the branch to the TAKE INVENTORY routine at 13100. To cover typing errors, INVENTORY has been conveniently shortened to INV. This means, of course, that if the player enters TAKE INVITATION he will find himself with an inventory in his lap instead, but there are no similar words to INVENTORY in the game vocabulary, so we needn't worry about this. If our player is that far out, he's never going to get very far as an adventurer anyway.

Starting with 5150 we see how the weight problem is handled. You'll remember that our variable for weight is W and that we decided to limit our puppet to carrying five items at any one time. So if W=5, whenever our player enters TAKE anything, he's going to get a report that he is carrying too much. Even if he says TAKE REST or TAKE ASPIRIN, he's going to see that he's carrying too much. The program will analyze taking ability only when his W has a value of less than 5.

In 5160 we skip to 5180, saving 5170 as the branch to go to whenever the player successfully takes a movable object. As you can see, this line increments W by 1, and it shuttles off to 50000 to redisplay the room, with the object just taken no longer displayed at the top of the screen. Line 5180 demonstrates a successful TAKE, reassigning a value of 1 to CO(1), meaning, of course, that this item is now a part of inventory and branching to 5170 for weight incrementation. Line 5190 is the logical exclusion from 5180, pointing out that the given movable object is not around at this time and scooting back to 3000.

You'll remember that to remove the crystal star in room 15 the player needed to use the odd piece of metal originally planted in room 19. No special manipulation is required; simply having it on the puppet's person would do the job. Line 5280 shows how this is translated into the program, accounting for all the various items necessary and acting accordingly. Line 5290 is the default when the star is not in room 15. Once the star has been loosened there's no need of the tool anymore, and the star can simply be picked up if it has been left behind. Line 5300 handles the situation when the player is in room 15 with the star in place but without the odd piece of metal.

There is no reason for the variations in lines 5350 and 5360, except I typed it two ways to alleviate the boredom of the same old thing over and over again. Once more we give the player the benefit of the typing doubt. The bunching up of robots in 5240 and the following lines is a result of our having two robots in the game (in addition to our puppet robot). There's the

destroyed one in room 5 and the attacking one in room 23. These lines account for both varieties and differentiate between the active and deactivated attacking robot in room 23. Line 5490 refers to a situation similar to the star/odd piece of metal before: This time we need a bucket before we can take any water. Line 5520 is a special cowcatcher telling the player that the puppet cannot take whatever it was he wanted taken.

Line 5530 is the point from which we save a game. Line 5540 is how you quit, allowing you both to quit for good and simply to quit and start over again. If the latter is the case, all the variables are cleared, and the program is routed back to the beginning, but keep in mind that the first few lines of the program are structured so the player can bypass all the introductory ballyhoo if he decides to play a saved game. Let's face it: How many times does any player want to sit through all that chatter from our puppet robot?

Line 5590 is wildly optional, but this is the sort of command most adventurers enter sooner or later.

Line 5610 introduces the subject of HELP. It is up to you as designer whether to provide player aids, and if so, how helpful those aids are going to be. As you can see, I have not included any in "Space Derelict!"; it's simply not that hard a game. Conversely, in a really hard game you might be defeating your purpose if you provide player help. But nonetheless if as a player you enjoy player aids, as a designer you might be inclined to put them in to suit your own tastes. The simplest way to handle a variety of HELPs—you'll probably want a number of them to cover a number of situations—is to organize them according to room number. So you might have a HELP routine something like this:

```
IF R = 1 THEN PRINT "MAYBE A MAGIC WORD MIGHT DO THE TRICK."
IF R = 5 THEN PRINT "LOOKED UNDER ANY ROCKS LATELY?"
IF R = 825 THEN PRINT "MANY A TEAR HAS TO FALL . . ."
PRINT "I'M AFRAID YOU'RE ON YOUR OWN HERE."
```

These lines would provide specific help for some of the rooms where you decide help might be necessary, and would simply flow through to the last line if the R value were not met, telling the player that help is not on its way, at least not in this instance.

Lines 5660–5670 are a variation on the star/pedestal business. This time we open a box and our puppet automatically reports an item in it if that item is there, and reports nothing if the box happens to be empty.

With a little bit of preplanning the section beginning at 5700 could have been reduced somewhat, but the way it's done shows the lazy man's approach, simply going room by room and answering the command OPEN DOOR.

Line 5910 opens one of the most amazing cans of worms, the FOR . . . NEXT stack. FOR . . . NEXT loops, as you know, require both halves to operate correctly, but this is more apparent when you have a NEXT without a FOR than the other way around because in the former case the computer comes right out and tells you so. Whenever the computer encounters a NEXT without a FOR the operation of the program is interrupted with a NEXT WITHOUT FOR error message. But the problem is just as crucial the other way around. You see, every time you enter a FOR command, the computer rings up one FOR on a stack it uses specifically for keeping track of such

things. Every time it encounters a NEXT it goes back and checks the FOR stack. So when a NEXT is encountered and there are no FORs on the stack, the computer knows right away that a problem exists and can tell you so immediately. But the computer checks the stack only when a NEXT is encountered, so a program with a lot of FORs and no NEXTs can run for quite some time—perhaps—without really screwing up the operation at hand. But sooner or later the stack will reach its limit—which happens to be 10 in Applesoft—and you've had it.

Let's look at some lines like this:

```
10 FOR X = 1 TO 100
20 IF L$(X) = "MORDRED" THEN GOTO 50
30 NEXT
40 GOTO 100
50 PRINT L$(X): GOTO 200
```

This is how the problem arises. We set up some criterion in the middle of a loop, and when this criterion is met, we exit the loop. This leaves the unmet NEXT command hanging, waiting to encounter some more Xs. And the tick on the FOR stack remains in place bcause the final NEXT (where X = 100)* has not been encountered: The computer has no idea that you're finished with this loop. This could affect the running of your program—and in such a subtle way that debugging can become a nightmare. At the very least the overloaded stack problem will eventually come up, and that will crash your program as sure as the day you were born.

So line 5910 is one way, albeit an inelegant one, around this problem. More attractive solutions are possible in some cases, and we'll be encountering them in our poker program.

Beginning at 6190 is the LOOK section. As far as I can tell I have provided a description of every object I have mentioned in every single room, plus one or two implied objects. I mentioned this before but I'll repeat it now: It is of the utmost importance that your program respond to mentions of everything that exists in your adventure. If you describe a room as having venetian blinds, even if they are purely decorative and unmovable, you must be able to respond to player input referring to those blinds. If a player is in a room with those blinds clearly described at the top of the screen, and GO BLINDS or TAKE BLINDS gets a blank response, then you've got a hole in your program. Successfully accounting for everything is sometimes an impossible feat—a stray item will occasionally get by the most diligent programmer, and even professional programs occasionally miss one or two pieces of the obvious furniture—but it is one you should try to accomplish at any cost.

We will zip now to line 6900, a variation on the theme of cowcatching,

*Actually, the final value of X—after line 30 in the example given—would be 101! NEXT statements increment your variable by the STEP value, measuring against the parameters you've established until the top number of the FOR . . . NEXT loop is reached. The loop in the example would keep incrementing X by 1 (the default STEP value). So if X = 1, when you hit the NEXT the program increments X by the STEP value, then compares X *minus* the STEP value to the top number set at the beginning of the loop. At the end X will equal 100 and once more the NEXT is encountered. Once again X will be incremented by the STEP and compared to the original parameter. In this case X will equal 101. Since 101 − 1 = 100, the FOR . . . NEXT loop will be terminated and processing will continue on the next line—and X will still be equal to 101.

where within a given S$ section (in this case LOOK) we respond to unidentifiable input by printing the S2$ rather than the whole C$. This clues the player into the fact that his first word was correct but that he might as well retire the second word as we just don't understand it.

Line 6920 and following cover DROP, the opposite of TAKE. It simply does everything the other way around, subtracting 1 from W rather than adding to it. And again, all successful DROPs will branch off to 50000 for a new room display, with the recently dropped movable object now added to the top of the screen.

And that's about it on the four-letter words. Hang in there, pilgrim, we're almost through it.

## THE REST OF THE CHITCHAT

This is the last batch of the conversation segment.

***"Space Derelict!"***

```
7210  REM  START 5 LETTER WORDS
7220  S$ = LEFT$ (C$,5)
    : IF LEN (C$) > 5 THEN S2$ = MID$ (C$,6
      )
7230  IF S$ < > "PRESS" THEN 7400
7240  IF R < > 2 THEN 7320
7250  IF S2$ = " BUTTON" THEN PRINT A$(9)
    : GOTO 3000
7260  IF S2$ = " BLACK" AND AL = 0 THEN AL =
      1
    : PRINT "THE AIRLOCK HAS CLOSED BEHIND
      ME."
    : GOTO 3000
7270  IF S2$ = " BLACK" AND AL = 1 THEN PRINT
      A$(21)
    : GOTO 3000
7280  IF S2$ = " BLUE" THEN 16000
7290  IF S2$ = " GREEN" AND D(10) = 0
      THEN D(10) = 1
    : PRINT "THE DOOR INTO THE SHIP HAS OPE
      NED."
    : GOTO 3000
7300  IF S2$ = " GREEN" AND D(10) = 1
      THEN PRINT A$(21)
    : GOTO 3000
7310  PRINT A$(1)
    : GOTO 3000
7320  IF R < > 7 THEN 7370
7330  IF S2$ = " BLUE" AND D(2) = 1 THEN D(
      2) = 0
    : PRINT "THE DOOR HAS CLOSED."
    : GOTO 3000
7340  IF S2$ = " BLACK" OR (S2$ = " BLUE" AND
      D(2) = 0) THEN PRINT A$(21)
    : GOTO 3000
7350  IF S2$ = " GREEN" THEN 16000
```

```
7360  PRINT A$(1)
    : GOTO 3000
7370  IF R = 24 THEN PRINT A$(21)
    : GOTO 3000
7375  IF R < > 15 THEN PRINT A$(7)
    : GOTO 3000
7380  IF LEFT$ (S2$,5) = " BUTT" OR S2$
        = " PANEL" THEN PRINT A$(21)
    : GOTO 3000
7390  PRINT A$(1)
    : GOTO 3000
7400  IF LEFT$ (C$,10) = "CLIMB WALL"
        THEN PRINT "IS THAT A METAPHORICAL
        STATEMENT? I      CANNOT CLIMB THESE
         WALLS."
    : GOTO 3000
7410  IF S$ = "TOUCH" THEN S$ = "PRESS"
    : GOTO 7230
7420  IF S$ = "CLIMB" AND R = 22 AND CO(12)
        = 0 THEN PRINT "OKAY."
    : FOR PAUSE = 1 TO 1000
    : NEXT
    : GOTO 16000
7430  IF S$ = "CLIMB" THEN PRINT A$(1)
    : GOTO 3000
7440  IF (C$ = "ENTER" OR C$ = "ENTER SHIP"
        ) AND R = 2 AND D(10) = 1 THEN R =
        3
    : GOTO 50000
7450  IF (C$ = "ENTER" OR C$ = "ENTER SHIP"
        ) AND R = 2 AND D(10) = 0 THEN
        PRINT A$(5)
    : GOTO 3000
7460  IF (C$ = "ENTER" OR C$ = "ENTER SHIP"
        ) AND R = 1 THEN R = 2
    : GOTO 50000
7470  IF C$ = "ENTER AIRLOCK" AND R = 1
        THEN R = 2
    : GOTO 50000
7480  IF C$ = "ENTER" OR C$ = "ENTER SHIP" OR
        C$ = "ENTER AIRLOCK" THEN PRINT A$(
        1)
    : GOTO 3000
7490  IF S$ < > "LIGHT" THEN 7530
7500  IF LEFT$ (C$,9) < > "LIGHT FLA"
        THEN PRINT A$(1)
    : GOTO 3000
7510  IF CO(8) < > 1 THEN PRINT A$(7)
    : GOTO 3000
7520  L = 1
    : PRINT "OKAY."
    : GOTO 50000
7530  IF C$ = "LEAVE" THEN PRINT "THERE'S N
        O TURNING BACK NOW."
    : GOTO 3000
7540  IF S$ = "LEAVE" THEN C$ = "DROP" + MID$
        (C$,6)
    : GOTO 6950
```

```
7550  IF C$ = "THROW" THEN PRINT A$(9)
    : GOTO 3000
7560  IF C$ = "THROW WATER" AND CO(15) = 1 AND
        R = 23 AND CO(13) = 123 THEN
        PRINT "IT STOPPED HIM DEAD IN HIS T
        RACKS."
    : CO(13) = 0
    : CO(16) = 123
    : GOTO 50000
7570  IF C$ = "THROW WATER" AND CO(15) = 1 THEN
        S$ = "LEAVE"
    : GOTO 7540
7575  IF S$ = "THROW" THEN PRINT A$(1)
    : GOTO 3000
7580  IF C$ = "SCORE" THEN PRINT "THE BUSIN
        ESS AT HAND IS THE SAVING OF   THE
        SOLAR SYSTEM. I WILL TELL YOU WHEN
        I HAVE SUCCESSFULLY COMPLETED THAT
        TASK."
    : GOTO 3000
7590  IF S$ = "LASSO" AND CO(12) = 1
        THEN PRINT "SERIES R ROBOTS ARE NOT
         QUALIFIED TO    PERFORM COWBOY CHO
        RES."
    : GOTO 3000
7600  IF S$ = "LASSO" THEN PRINT A$(8)
    : GOTO 3000
7610  IF S$ = "BLAST" THEN S$ = "SHOOT"
    : GOTO 7650
7620  IF C$ = "THINK" THEN PRINT "THINK WHA
        T?"
    : GOTO 3000
7630  IF S$ = "THINK" THEN C$ = "SAY" + MID$
        (C$,6)
    : GOTO 4750
7640  IF S$ < > "SHOOT" THEN 7720
7650  IF C$ = "SHOOT" THEN PRINT A$(9)
    : GOTO 3000
7660  IF CO(7) < > 1 THEN PRINT "I AM NOT C
        ARRYING MY LASER PISTOL."
    : GOTO 3000
7670  IF S2$ = " ROBOT" AND CO(13) = 100 +
        R THEN PRINT "IT DOESN'T HAVE ANY E
        FFECT! IT DOESN'T  EVEN SLOW HIM DO
        WN!"
    : GOTO 3000
7680  IF S2$ = " ICE" AND R = 22 THEN
        PRINT "OKAY."
    : CO(14) = 100 + R
    : GOTO 50000
7690  IF S2$ = " DOOR" AND R = 23 AND D(9) =
        0 AND CO(16) = 123 THEN PRINT "IT W
        ORKS. I CAN GET THROUGH NOW."
    : D(9) = 1
    : GOTO 3000
7700  IF S2$ = " DOOR" AND R = 23 AND CO(16
        ) = 0 THEN 16000
```

```
7710  PRINT "SERIES R ROBOTS ARE DESIGNED
       WITH MAXI- MUM SAFETY IN MIND. BECAU
       SE OF THIS I   CANNOT " + C$
    : GOTO 3000
7720  IF C$ = "WHERE" THEN PRINT A$(1)
    : GOTO 3000
7730  IF S$ = "WHERE" THEN PRINT A$(7)
    : GOTO 3000
7740  IF (C$ = "CLICK" OR C$ = "CLICK PENTA
       GON") AND CO(2) < > 1 THEN
       PRINT "I'LL HAVE TO HAVE IT IN MY H
       AND FIRST."
    : GOTO 3000
7750  IF C$ < > "CLICK" AND C$ < > "CLICK P
       ENTAGON" THEN 7900
7760  IF R = 3 AND D(12) = 0 THEN D(12)
       = 1
    : PRINT "THE WEST DOOR HAS OPENED."
    : GOTO 3000
7770  IF R = 6 AND D(2) = 0 THEN D(2) = 1
    : PRINT "THE SOUTH DOOR HAS OPENED."
    : GOTO 3000
7780  IF R = 18 AND D(5) = 0 THEN D(5) = 1
    : PRINT "THE NORTH DOOR HAS OPENED."
    : GOTO 3000
7790  IF R = 21 AND D(7) = 1 AND D(5) = 1 THEN
       PRINT A$(21)
7800  IF R = 21 AND D(5) = 0 THEN D(5) = 1
    : PRINT "THE SOUTH DOOR HAS OPENED."
7810  IF R = 21 AND D(7) = 0 THEN D(7) = 1
    : PRINT "THE NORTH DOOR HAS OPENED."
7820  IF R = 21 THEN 3000
7830  IF R = 22 AND D(8) = 0 THEN D(8) = 1
    : PRINT " THE NORTH DOOR HAS OPENED."
    : GOTO 3000
7840  IF R = 23 AND D(8) = 1 AND D(7) = 1 THEN
       PRINT A$(21)
7850  IF R = 23 AND D(8) = 0 THEN D(8) = 1
    : PRINT "THE SOUTHERN DOOR TO THE WEST
       HAS OPENED"
7860  IF R = 23 AND D(7) = 0 THEN D(7) = 1
    : PRINT "THE SOUTHERN DOOR TO THE EAST
       HASOPENED."
7870  IF R = 23 THEN 3000
7880  IF R = 10 THEN 16000
7890  PRINT A$(21)
    : GOTO 3000
7900  PRINT A$(2) + C$
    : GOTO 3000
7910  REM      START 6 LETTER WORDS HERE
7920  S$ = LEFT$ (C$,6)
    : IF LEN (C$) > 6 THEN S2$ = MID$ (C$,7
       )
7930  IF C$ = "INSERT" THEN PRINT A$(9)
    : GOTO 3000
7940  IF S$ < > "INSERT" THEN 8050
7950  IF R = 24 THEN 8001
```

```
7960  IF CO(5) < > 1 THEN PRINT "I HAVE NOT
        HING TO INSERT IT IN THAT I   CAN S
        EE."
    : GOTO 3000
7970  IF S2$ = " METAL" OR S2$ = " PENTAGON
        " OR S2$ = " WEAPON" THEN PRINT A$(
        1)
    : GOTO 3000
7980  IF S2$ = " CRYSTAL" AND CO(3) = 1
        THEN 16000
7990  IF S2$ = " STAR" AND CO(6) = 1
        THEN CO(6) = 5
    : PRINT "IT FITS LIKE A GLOVE. I THINK
        THAT WAS  A GOOD IDEA, BUT I DON'T
        KNOW WHY."
    : W = W - 1
    : GOTO 3000
8000  PRINT A$(1)
    : GOTO 3000
8001  IF S2$ = " BOX" THEN 8010
8002  IF S2$ = " STAR" AND CO(6) = 1 AND CO
        (5) = 1 THEN PRINT "I'LL PUT IT INT
        O THE BOX... IT FITS     LIKE A GLO
        VE. I THINK THAT WAS A GOOD   IDEA.
        "
    : W = W - 1
    : CO(6) = 5
    : GOTO 50000
8003  PRINT A$(1)
    : GOTO 3000
8010  IF R = 24 AND CO(6) = 5 AND CO(5)
        = 1 THEN 17000
8020  IF R = 24 AND CO(5) = 1 THEN 16000
8030  IF R = 24 THEN PRINT A$(7)
    : GOTO 3000
8040  PRINT A$(1)
    : GOTO 3000
8050  IF S$ = "UNLOCK" THEN PRINT A$(8)
    : GOTO 3000
8060  IF C$ = "REMOVE" THEN PRINT A$(9)
    : GOTO 3000
8070  IF C$ = "REMOVE STAR" AND CO(6) = 5 THEN
        CO(6) = 100 + R
    : GOTO 50000
8080  IF LEFT$ (C$,6) = "REMOVE" THEN
        PRINT A$(1)
    : GOTO 3000
8090  PRINT A$(2) + C$
    : GOTO 3000
8100  S$ = LEFT$ (C$,7)
8110  IF LEN (C$) > 7 THEN S2$ = MID$ (C$,8
        )
8120  IF S$ < > "UNLIGHT" THEN 8160
8130  IF (C$ = "UNLIGHT" OR LEFT$ (C$,12) =
        "UNLIGHT FLAS") AND L = 1 THEN
        PRINT "OKAY"
    : L = 0
    : GOTO 50000
```

```
8140  IF (C$ = "UNLIGHT" OR LEFT$ (C$,12) =
        "UNLIGHT FLAS") AND L = 0 THEN
        PRINT "THE FLASHLIGHT IS NOT LIT."
    : GOTO 3000
8150  PRINT A$(1)
    : GOTO 3000
8160  IF C$ = "EXAMINE" THEN PRINT A$(9)
    : GOTO 3000
8170  IF S$ = "EXAMINE" THEN S$ = "LOOK"
    : GOTO 6210
8180  PRINT A$(2) + C$
    : GOTO 3000
8190  IF C$ = "DESCRIBE" THEN PRINT A$(9)
    : GOTO 3000
8200  IF LEFT$ (C$,8) = "DESCRIBE" THEN S2$
        = MID$ (C$,9)
8210  IF S2$ = " AIRLOCK" OR S2$ = " ENTRAN
        CE" THEN PRINT A$(22)
    : GOTO 3000
8220  IF LEFT$ (C$,8) = "DESCRIBE" THEN S$ =
        "LOOK"
    : GOTO 6210
8230  PRINT A$(2) + C$
    : GOTO 3000
8240  IF LEFT$ (C$,5) = "INVEN" THEN 13100
8250  PRINT A$(2) + C$
    : GOTO 3000
```

We've covered almost every possible programming variation for this segment now, so most of this is just more of the same. But there are a couple of new twists. Line 7220, for example, finally puts all the S$/S2$ business on one line, making it as elegant and concise as possible. Now that you know what this is all about, this is the form you should use for every length variation.

The PRESS section covers all those buttons in rooms 2, 7, and 15 (with the synonyms TOUCH and PUSH also bringing us to this section). In 7440 you encounter ENTER, which is used as a substitute for GO at the beginning of the game where no EXIT directions are given. Covering such input as GO IN or ENTER in less demanding situations is entirely optional, but it is recommended in likely cases. Why shouldn't GO IN be a likely synonym for GO DOOR? The former is certainly more colloquial than the latter, at least in the real world, while the latter has its place really only in adventure lingo. While the designer must cover all the normal adventuring usages, there's nothing wrong with extending yourself as much as possible when there are perfectly good English phrases that can do what the perfectly good computer phrases also do. It's the old user-friendly concept; of course, user-friendly usually translates as "any idiot can understand it," but what's wrong with that? Accessibility is the name of the game, and you're not going to get very far as a game designer if you're the only one who can communicate with your programs.

Line 7580 brings up the business of providing a SCORE for the player as

he progresses. In a situational adventure like "Space Derelict!" scoring is a non sequitur, since either you've defused the ship or you haven't. But in treasure-hunt adventures the score can keep the player apprised of how many treasures he's already collected vs. the total number of treasures he needs to win. There are two usual ways of handling this. The first is giving the player an artificial score from 0 to 100 based on the number of treasures he's already collected, simply by dividing the number of treasures he already has by the total number of treasures and multiplying the result by 100. That way, if there are 20 treasures, for instance, he would get five points for each one. A new variable such as TR would handle this for you:

```
IF C$ = "SCORE" THEN PRINT "OUT OF A POSSIBLE 100 YOU HAVE ";INT( (TR/20) * 100)
```

Of course, TR is the running count of the number of treasures collected, and in this hypothetical case 20 is the total number of treasures. The other way to handle it, again using TR as the variable count of collected treasures, is simply to say:

```
IF C$ = "SCORE" THEN PRINT "OUT OF A POSSIBLE 20 TREASURES YOU HAVE ";TR
```

Whichever suits your fancy is perfectly appropriate. The value of TR would be established by totaling the number of treasures on hand in the treasure-storing room.

THINK in 7620 is the introduction of the telepathy factor, a very strange word indeed for an adventure. Line 7710 is a good example of a programming shortcut. As a rule, if you give your puppet a gun you have to accommodate his shooting everything in sight and would normally need a routine as long and complicated as the LOOKs were earlier. But by putting a little common sense into the robot's head we're able to bypass all the shoot-'em-up possibilities by allowing only for the shots fired that we wish to be fired and circumventing the rest by telling the player that the robot has more self-control than he does.

Lines 7720 and 7730 are not at all necessary for practiced adventurers, but I've never seen a newcomer who didn't try to get the computer to do the work of finding things for him by saying WHERE (WHATEVER) as if that would solve all his problems. So our user-friendly program includes it here. Line 8120 introduces the hideous word UNLIGHT, which is the only one I've heard of that will turn off the lantern or candle or whatnot in just one word. I'd really like to know if there are any better ones around. Line 8240 allows the bad typist to enter any variation he likes (provided the first five letters are correct) of the word INVENTORY to get same. And for those of you who never thought you'd see the end of this, line 8250 is the end of the conversation-with-the computer segment.

## MISCELLANEOUS BUSINESS

We've now gone through 99.9 percent of all the programming necessary for a standard situational-type adventure, with a few side roads mentioned occasionally for the other kinds of adventures you might want to design. A few more tidbits like these will be discussed in Chapter 6, providing you with a basis for just about anything you might want to put into your games. The

important point to keep in mind is one that was mentioned earlier and should become your designing motto: Almost everything can fit into the normal operation of the basic adventuring module, and one way or another you should *make* it fit. As you've seen in "Space Derelict!" we've handled everything from simple DROPs and TAKEs to INSERT, THROW, and BLAST, all in the conversation segment. But all these actions had one thing in common: They were all taken in response to input from the player. None of them was self-generating, which leaves us with the problem of finding a place to put those situations within the game where something will happen out of the player's control. In "Space Derelict!" this situation arises only once, when the player enters room 23, where the killer robot is waiting to pounce. Unless the player deactivates this robot or immediately leaves the room, the puppet robot will be destroyed after three turns. This is not as elementary as the trap in room 5, where all the player had to do was walk into the room to get zapped, and which was concisely handled in the GOs. This time the player has three moves before he enters Zap City, and we must accommodate him accordingly. Any miscellaneous business you come up with will be of a similar nature, be it roving monsters, a built-in clock limiting player moves, or whatever. You want to insinuate this material into the program so that it comes up only when you want it to, the way you want it to. The basic module gives you both a place to account for such activities and a place to program their details.

The place to account for such activities is at the beginning of the line 3000 conversation-with-the-computer segment, *before* prompting the player input. Go back and look at lines 3000–3020. Line 3000 is just our REM, and 3020 is the player prompt. It's line 3010 that handles the miscellaneous business in this game, and any number of other miscellaneous businesses also could have been inserted at this point, giving the necessary criteria and, if the variables fell within that criteria, branching out somewhere to handle the appropriate business. The reason this spot works so well for this is that all the subroutines do eventually branch back to 3000 and the player prompt, be they changes of room in 50000, or be they any actions covered in the conversation-with-the-computer section itself. All roads lead to 3000, so 3000 is the place for miscellany.

In "Space Derelict!" the criterion for getting involved in our miscellaneous business is simply our player being in room 23, where he would encounter the killer robot, so all line 3010 does is branch to 40000 on the condition that R=23. If R does not equal 23, then the program continues along to the input prompt, completely unaffected. If R does equal 23, we hit the final programming lines of "Space Derelict!"

***"Space Derelict!"***

```
40000 REM  ROBOT
40010 IF CO(13) = 0 THEN RETURN
40020 T = T + 1
40030 IF T = 1 THEN RETURN
40040 IF T = 2 THEN PRINT "THE ROBOT IS ATT
        ACKING ME!"
    : RETURN
40050 IF T = 3 THEN 16000
```

This short subroutine handles the attack of the killer robot. You'll recall the unexplained T=0 statement in line 50030, the outset of the room segment. This simply cleared the value of T for us so it wouldn't be hanging over our robot's head when he was out of room 23. T (for "time") is merely our counter for the number of moves to be taken in room 23. When the player enters room 23 and CO(13), the killer robot, is alive and well, the value of T is set at 1 by line 40020 (since T=0 to begin with, T+1=1). Then line 40030 returns us to 3010 and the player prompt. The player knows the killer robot is in the room, of course, because it will be displayed at the top of the screen in the room description.

Whatever move the player now makes—other than leaving room 23, which would get him out of the situation entirely—will bring him back to the beginning of 3000 and another run through the 40000 subroutine. Once again T will be incremented by 1, but this time T+1=2, so line 40040 comes into play, and the puppet reports that the killer robot is at his throat, and the play branches back to the player prompt. Again, no matter what the player does, short of leaving room 23—or destroying the killer robot—he will branch back here again, and this time T will equal 3, meaning hello line 16000 and good-bye player. Once the killer robot is deactivated, however, all of this becomes moot per line 40010.

There is one hole in the way this is handled here, and it could have been sealed up, but if the player discovers it, it could make things more interesting for him. If you've figured out what that hole is, go to the head of the class. You see, if the player enters a successful DROP or TAKE command while he is under siege by the killer robot, the program will branch to 50000, resetting T at 0! So if the player wants to stand there in room 23 and keep picking up and dropping things, he can do so until kingdom come and not get killed. But he won't progress any further in the game because the door to room 24 is sealed in more ways than one. That is, the player can't pull this trick to buy himself some extra time to blast that door, because there is a built-in fail-safe against such actions—go back and take a look at 7700.

There is plenty of room between lines 40000 and 50000 to fit in any kind of miscellaneous business you want, but as you can see, there is one drawback to it. Every piece of miscellaneous business requires extra (and probably complicated) programming for successful execution. Everything else in the program works smooth as a goat, whereas treading into the waters of miscellany and 40000 can lead to entirely new problems. But not every complication is going to fit neatly into the prescribed pattern, so we have to have some place for the squeakers that just don't belong anywhere else, and at least now you know.

And believe it or not, we have now covered all of the "Space Derelict!" program, so if you have been entering it as it's been printed in the book, you now have the whole thing up and running—provided you've typed it all correctly. For the most part you now know everything you need to know to set out on your own adventure design. But writing your own adventures is not quite as easy as copying someone else's, so we have a little more ground to cover in the next couple of chapters.

# 5

# BUGGING AND DEBUGGING

## A RECAP OF THE ADVENTURE MODULE

Now we've gone bit by bit through an entire adventure, covering everything from the basic concept of what adventuring is all about, through the theories of programming techniques, right down to the nitty-gritty of the actual line programming. You should be ready to attack your own design at this point with a fair certainty of a reasonably running program when you're finished. But in this chapter I want to talk about something nobody ever seems to cover. When you read articles or books on computers, everybody tells you how to do everything right, showing you perfect programs and telling you how they run. But nobody ever tells you how that program got to be so perfect in the first place. How many drafts did that little ten-liner go through to get where it is today? By the time something gets to the publishing stage, not only is it debugged cleaner than a clerical whistle, but it's usually so polished that the *running* of the program becomes the end-all and be-all, not the understanding of *how* the program works. In many cases this makes for completely unintelligible—but nonetheless perfect—programs. No one will ever accuse us of that in "Space Derelict!" If you have a good eye for program stylization and streamlining (topics covered in Part Two of this book) you could probably shave 20 percent off the top of that one with no trouble at all. Rewrite large portions of it with all the action translated into variables and you could get out as much as 50 percent. But, of course, the program would lose much of its instructional benefit, which was the whole point of the exercise in the first place. Anyhow, what I want to discuss in this chapter is how "Space Derelict!" got to where it is vis-à-vis the actual entry process. There's more to writing a program than just writing a program, especially with adventures where we're dealing with a basic module

readjusted each time out to fit each new game. The module will remain the same, as will the programming and editing techniques, and these things are what we'll be talking about here.

First let's abstract the adventure module in a usable form so that you will have a basic outline for your own designs. This is the framework on which you and your adventures will hang (or be hanged, depending on the circumstances).

```
10 SET DIMENSIONS
20 SET MORE DIMENSIONS
30 CLEAR SCREEN
40 GOSUB 15000: REM INTRODUCTION
50 CLEAR SCREEN
60 GOSUB 2000: REM INITIALIZATION
80–110 ALLOW FOR REPLAY OF SAVED GAME
120 CLEAR SCREEN
130–300ish CONTINUE INTRODUCTION AS NECESSARY
310 GOTO 50000: REM ROOM ROUTINE
```

---

```
2000 INITIALIZATION/HOUSEKEEPING
2010 R = 1
2020–2190ish SET A$(X): REM ANSWERS FROM THE COMPUTER
2200–2490ish SET CO$(X) AND CO(X): REM MOVABLE OBJECTS
2500–2690 SET MISCELLANEOUS VARIABLES
2700 RETURN
```

---

```
3000 CONVERSATION-WITH-THE-COMPUTER SEGMENT
3010–3050 SET MISCELLANEOUS BUSINESS PARAMETERS AND
  GOSUBS
3060 INPUT "PLAYER PROMPT";C$
3070–3200ish BREAK DOWN C$ BY SIZE AND GOTO APPROPRIATE
  GROUP
3210–wherever SET ACTUAL CONVERSATION
```

---

```
13000–13040 SET R$ SUBROUTINE
```

---

```
13100–13150 SET INVENTORY SUBROUTINE
```

---

```
15000–15200ish SET INTRODUCTION SUBROUTINE
```

---

```
16000 LOSING ROUTINE
16010–16100 TELL PLAYER HE HAS LOST IN NO UNCERTAIN
TERMS
16110–16140 GIVE PLAYER OPTION TO TRY AGAIN
16150 IF PLAYER SAYS YES, CLEAR AND GOTO 10
```

---

```
17000 WINNING ROUTINE
17010–17200 CONGRATULATE PLAYER FOR A JOB WELL DONE
17210 END
```

---

30000 GAME-SAVING SUBROUTINE
30010 SET D$ AS CHR$(4) (CTRL-D)
30020–30050 OPEN, DELETE, OPEN, AND WRITE GAME-SAVE FILE
30060–30200ish PRINT ALL GAME VARIABLES INTO GAME-SAVE FILE AND CLOSE GAME-SAVE FILE
30210 GOTO 3000

---

31000 REPLAY SAVED-GAME SUBROUTINE
31010 SET D$ = CHR$(4) (CTRL-D)
31020–31030 OPEN AND READ GAME-SAVE FILE
31040–31200 ish INPUT ALL GAME VARIABLES AND CLOSE GAME-SAVE FILE
31210 RETURN

---

40000–49999 MISCELLANEOUS BUSINESS SUBROUTINES AS NEEDED

---

50000 ROOM LAYOUT ROUTINE
50010 GOSUB 13000: REM GET R$
50020 POKE 34,0: POKE 35,13: HOME: REM TEXT WINDOW
50030 and further MISCELLANEOUS BUSINESS FACTORS AS NECESSARY
50040 ON R GOTO
50050–52000ish ROOM DESCRIPTIONS
52010 POKE 35,24: POKE 34,13: REM TEXT WINDOW
52020 HTAB 1: VTAB 24: REM SET CURSOR AT BOTTOM OF SCREEN
52030 GOTO 3000

---

63000 FOR PAUSE = 1 TO 2500: NEXT: RETURN

And that's the whole ball game in its briefest form, with a lot of those numbers, of course, varying as the need arises.

## GETTING IT INTO WORKABLE SHAPE

The nice thing about adventures is the comparative ease of programming. The not-nice thing about adventures is that each one is unique, with very few pieces that can be transported 100 percent intact from one adventure to the next. A few pieces can travel without variation, but for the most part everything depends on the demands of your game universe—and your game universe changes from adventure to adventure (or else you've taken up the wrong hobby). With strategy games this is not necessarily the case. Compare bridge, poker, and gin rummy, for example; they may be radically different in many ways, but at least they all revolve around cards, and programs for these games can contain identical shuffling routines, plus only slightly modified dealing routines and (if you use them) graphics. Another strike against adventures in pure programming terms is their lack of structure. For the most part all we're doing is going around and around the line 3000 conversation-

with-the-computer segment, with occasional side trips to 50000 or 13000 or 13100. What we have here is more an extended loop than a program, and this does restrict us somewhat in that we can't take the program apart and analyze it bit by bit in our debugging stage; we're stuck with one enormous monster to be attacked *in toto*.

This means that you can't begin any serious debugging until you've finished the entire program. The point of this section is to provide you with some program-writing techniques that may save time when you reach the other end of the rainbow and have to start making your brainchild run. And the first rule of programming is one I'm sure you've heard over and over again: SAVE your work often and well. If you have to get up to go to the bathroom, SAVE what you've done so far. If there's a fire in the basement, SAVE your work before you grab the extinguisher. If the nuclear plant next store seems to be giving off a strange purple glow, SAVE your program before going to the window to get a better look. If you live alone in a garret this may not be much of a problem, but if any animal, human or otherwise, young or old, shares your living quarters with you and your computer, sooner or later he, she, or it is going to discover some unique and amazing way to DELETE seven hours of your hard work without even realizing it. A non-human problem that plagued me originally when I acquired my computer was the phantom power surge. I'd be writing away like a bandit and the next thing you know I'd have an empty memory (and so would my Apple). I have no clear technical idea what power surges are, but they seem to suck the juice right out of your computer's memory banks and ship it down the river. However, you can purchase a little device called a surge protector for $60 or $70 and it's well worth the investment. I haven't lost a syllable since I acquired mine.

The next thing to get your hands on is the little program known as Renumber, which came with your System Master disk. If for any reason you did not get Renumber, or if you've lost it or the dog chewed it up, get your hands on a copy quick from any other Apple owner.* Renumber does exactly what it says it does, and its value is enormous. It allows us to adjust programs to let more air between the lines, and it allows the merging of programs, which also has its benefits. When I program and debug I find Renumber one of my most important tools, allowing me more room when I need it and closing up the gaps behind me when I don't. There's something about the continual neatness of 10 increment lines that just keeps me going when the going gets tough: Renumber keeps me in 10-increment heaven.

There are various programs called line editors available on the market that work much like word processors. These are quite expensive, and while they look very nice, you may not consider them worth the price for the service rendered. Among other things, they can help you find and change items in a program almost instantaneously (much in the same way as a word-processing program does a search). Many programmers swear by them, so you might want to take a closer look. Also, some word-processing programs will accept modified Basic program files, allowing you with some maneuvering to ma-

*A good renumber program is included on the Double-Take disk used to print the programs in this book, but I must admit that I did not use it when debugging "Space Derelict!" (it hadn't come out on the market yet). The Apple program worked just fine.

nipulate your programs as if they were created in your word processor. Mine doesn't, but read the fine print on yours—you might find this capability handy.

The first step in the programming procedure is to RUN Renumber. This is a machine-language program neatly tucked away into your computer memory in a quiet corner untouched by any other programming you'll be doing in your adventure. When you need it it will be there, but it will remain completely unobtrusive until that time. Now we can get down to business.

I have mentioned before various bits and pieces of the entry process, so bear with me if I repeat myself here. The first thing to do—after you've designed your game and made your map and run Renumber—is enter the two short routines at 13000 and 13100. At this stage you should know how many movable objects you have, and you can therefore enter the correct numbers into these routines where they are called for. If for any reason the number of movable objects changes along the way (and this isn't as unlikely as it sounds), don't forget to come back here and revise these numbers.

Next, start on the 50000 room-layout routine. It is not a bad idea to leave a little space—four or five program lines, say (I'm assuming here that you're entering your statements in the normal increment of 10)—at the beginning before the ON R GOTO line for any miscellaneous business that may come back to haunt you later. Then do the room layouts as mentioned earlier, augmenting the ON statement as you go, discovering exactly where the first lines of each room will be. And as you go along REM your routine exit GOTO statements until the end line is known, filling them in later. Obviously you can't know where to branch until you've come to the end of your room descriptions. Once you know where your exit from the routine will be, these lines will all branch there.

You'll notice some slight variations in the number of blank lines (the empty PRINT statements) in the room-layout routine. These changes came in the debugging stage. Originally every line of text was separated by one blank PRINT line. It's best to design these room layouts in whatever way is pleasing to your eye, and if one or two of them don't fit because they're too long and the accumulation of R$ objects during the actual play throws them haywire, you can always come back later and shave them down to size.

After entering all the descriptions and going back and setting the ON R GOTO and the exit GOTOs from all the rooms, you're ready to SAVE all this and, if you wish, print it. As I recommended earlier, you might find it best to work from a printout of the room-layout routine.

If you're going to use a pause, enter it now. I prefer the upper reaches because the whole point of a pause is to slow something down, and what could be slower than dredging through a whole program to find line 63000? Must take a whole microsecond, give or take a fraction or two.

My next step (by the way, although this sounds as though I'm sitting there zipping through all this, we're actually talking about quite a few hours of work here) is to program the starting lines of the game. At this point you may be a bit fuzzy about how large an array to set for A$ and perhaps even the CO family, so do the best you can and keep in mind that these lines may need changing later. Some of this section will vary from game to game, while some is fixed, such as the branches to 15000 and 2000 and the replay-saved-game prompt. As you continue the introduction you'll be doing some

actual creative composition, and you might want to work some of this out on paper first if you feel more relaxed writing that way. Whichever you prefer, writing at the keyboard or on good old-fashioned paper, when you've finished entering the entire introduction you are now going to call on Renumber. Let's say your introduction stretches from line 10 to line 250. Simply type in

& FIRST 10, START 10, END 250, INC 20

and Renumber will automatically renumber the lines from 10 to 250 in increments of 20 rather than the increments of 10 (if this command looks like so much gibberish, fear not, because there are complete, easy-to-understand instructions right at the beginning of Renumber when you run it). You could, of course, number in increments of 20 in the first place, but you get so used to increments of 10 that it's hard to think in terms of anything else. The reason for allowing so much space between lines is simple: We're preparing now for the errors to come, allowing space for addenda that will no doubt be necessary as we start our debugging/revising frenzy. It might not be absolutely essential in some of the sections of the program, but when we get to the conversation-with-the-computer segment this extra space will be our guiding light, our prayer, and our salvation. And it doesn't hurt to leave space in the other sections as well where there's a possibility it might come in handy. We didn't do it in the room layouts because that section tends to get smaller with editing rather than larger. Not so with the introduction.

The next step is the housekeeping at 2000. Here we're going to find ourselves quite tongue-tied at the first run-through. We'll have some A$ answers and most of the CO$ movable objects, but every time we turn around in the line 3000 routine we'll find ourselves needing another A$ and perhaps even more CO$s for spice, so numbering the 2000 segment in increments of 20 is recommended. As the program progresses and you come up with new A$s, simply tack on each one after the last one here and go back and update the DIM in the first lines. "Space Derelict!" began with A$(1) through A$(7)—the rest were added as the situations arose. The same is true of the miscellaneous variables at the end of this section. It's almost impossible to know what these are before you create them, and you'll find yourself coming back and augmenting this list throughout the rest of the programming.

Keeping in mind the fluctuation of variables, now you can enter the game-saving and replay routines starting at 30000. And remember: Every time you change a program variable anywhere else, *change it here too.*

Next you have the winning and losing routines, which aren't too hard to get right the first time. And then your last easy job—except that it's off-the-cuff programming that will require more work and thought than all the rest of the programming combined—is the miscellaneous business at 40000. Do your best and hope for the best, in that order.

Now we get to the crusher, the conversation segment at 3000. There's no way around this one, and the best thing to do is take it one word at a time. Try to think of every factor you can pertinent to that word and how it can change in every room, and then go on to the next word. No matter how much you do think of, you'll always leave out about half of it, so this is where line increments of 20 really come in handy. Using "Space Derelict!" as a guide should help quite a bit in covering a lot of the possibilities, but only in a general sense, as each new adventure is going to be radically different from

every other. When you actually start running the program, you'll be amazed at all the mistakes you made, no matter how well you think you got it down the first time.

The final step in the programming is to make a printout of the entire adventure. This time the hard copy of the program is a must, because it is well nigh impossible to edit a program this long if you don't have it laid out on paper right in front of you. Also, the printout will let you carry the program and tinker with it anywhere you want—at work or at school or riding the train or walking the dog—all those places you can't take your computer. And now, with your printout in hand and a fully written adventure on your disk, you're ready to start debugging.

## EDITING TECHNIQUES

The more complicated the program, the more problems there will be. For me there are two points in the creation of any new long program where I feel like throwing in the towel and starting something new. The first of these is in the middle of the actual entry of the program. By this time I know exactly where everything is going and I'm right in the middle of what seems to be endless typing. There is nothing creative about this part of the process, and I always wish I could hire somebody to do it for me. The second stumbling block is the debugging. You never know what you're going to find when you start unwrapping your program and trying to get it to run. Sometimes you'll find baffling problems where the program will seem to go off on some disastrous tangent that simply makes no sense. At other times you'll get syntax errors in lines that look completely correct, meaning you've made some mistake you didn't even know *was* a mistake. But one way or another you've got to solve these problems, and that's what we'll talk about now.

The first step in debugging is to take your printout and go over the program on paper. You won't find many logic errors at this stage, especially in an adventure where the logic is so simple. But you will spot some syntax errors, which are usually no more than typographical errors you made at the entry stage. My most common error is omitting the question mark when I type in PRINT statements (the computer, of course, translates the ? into a PRINT). So instead of having lines that read:

110 PRINT "YOUR NAME AND ADDRESS"

I'll end up with:

110 "YOUR NAME AND ADDRESS"

These are easy to spot and easy to fix. Punctuation is always a trouble spot, especially when you're concatenating with semicolons, as in:

666 INPUT "WHAT'S YOUR SIGN?"; SIGN$

It's easy as pie to put the semicolon before the quotation mark, or the question mark after the quote. But this is the sort of thing you can pick up pretty easily simply by proofreading. My practice is to run through and mark all the changes on the printout, then LOAD the program and make the changes, checking off each change on the printout as I go. This way I still have an up-to-date printout, edited to match the actual program.

The next step is to RUN the program and see what happens. With luck you might actually get into the first room, but I wouldn't count on it. First

of all, the odds are that you'll encounter spelling errors or bad line breaks in the introduction that will have to be fixed. And more than likely the computer will spit back at least one syntax error at you. And if life is really against you, you might even branch into some no-man's-land you can't find any way out of. The first problems can be fixed easily enough, but crazy branches are another thing altogether. They require use of the TRACE command.

TRACE is the programmer's lifeline. No matter how complicated your program logic, TRACE can follow it. All you have to do is track your TRACEs and you're home free. But it's not always as easy as it sounds.

First of all, TRACE can be used either as a direct command, as in:

```
TRACE
```

or as a statement within a program, as in:

```
10 TRACE
```

When you TRACE a program, the line number of every statement processed is sent to the screen. If a line contains two or more statements separated by colons, each statement processed will get a repeat of the line number. So if you have:

```
1040 PRINT "BUY WAR BONDS!": X = 1943: Z$(5) = "U.S.O."
```

and you TRACE, you will see the following on the screen:

```
1040 BUY WAR BONDS
1040 1040
```

Since 1040 goes through the wringer three times, you TRACE three times. If there was some reason not to process those continuing colon statements, this would not be the case. If we had:

```
1030 I = 19
1040 IF I < > 19 THEN PRINT "BUY WAR BONDS": X = 1943: Z$(5)
= "U.S.O."
1050 END
```

and TRACEd we'd get:

```
1030 1040 1050
```

The rest of 1040 would not be traced, since that information was not processed as a result of I being equal to 19. The computer does evaluate the first part of this IF . . . THEN but goes no further (and takes no action) when the criterion is not met.

The key to the successful use of TRACE is not to go overboard with it. When you use it as a direct command you end up TRACEing *everything,* and usually this is not necessary. A program will run correctly for a while and then go wacko. Find the last place in the program where everything was okay and insert the TRACE there in the actual program. That way everything will run without the program lines where they are unnecessary (TRACE can make a video display look like a number factory), and will start reporting those lines to you only on or near the trouble area. Keeping TRACE contained like this makes it a lot easier to handle.

Your biggest use of TRACE in an adventure will be in the conversation segment, but you may find yourself using it elsewhere as well. When all else fails, keep it in mind. By the way, NOTRACE is the command to deactivate TRACE.

Once you've gotten past the introduction in your debugging, you'll find yourself standing at the threshold of your first room. The first command you should enter is SAVE GAME, for two reasons. First, you can see if your

saving routine is oiled and running, and second, you won't have to go through the introduction again if you break into the program. It's a good idea to do this as you enter every room. In a process like debugging where there are few time-savers, you should make the most of this one. When you save the game and everything is hunky-dory, the little red light will come on on your disk drive and without further ado will go off a couple of seconds later, and you will get a new WHAT DO YOU WANT TO DO? prompt. If this is the first time you've tried creating this particular game-save file, you might get a FILE NOT FOUND response from the operating system when it hits the first OPEN statement preceding the DELETE statement in the saving routine. If this happens, try RUNning the program beginning at the *second* OPEN statement following the DELETE command (in "Space Derelict!" this would mean "RUN 30040"). This will get the game-save file on the disk, giving DOS something to DELETE when the time comes. Next time through, it should work like a charm.

After saving a game, quit and start again. This time when you get to the choice of whether or not to play a saved game in the introduction, say yes and see what happens. You should find yourself back in room 1, period. There is a chance, however, that you'll first see the DOS error notice EXTRA IGNORED, perhaps more than once, as the game-save file is being accessed. If so, there is a discrepancy between the number of items in the save routine and the number in the replay routine. Compare the two and fix the error.

Now you're back at the threshold of room 1, and SAVE GAME has been cleaned up. Next, make sure that all the exits are as they're supposed to be. Let's say you have a north exit and a south exit. First enter N. You should find yourself in the room to the north. Go back to room 1 and try S. This time you should move south. Now go back and try the incorrect W and E. These should *not* work and should bring to the monitor the appropriate no-can-do response (the same for U and D, if you're using them). Make sure this is all working before proceeding further.

Next, take a look at the top of the screen, the room description. What you want to do now is make sure you've accounted for everything you've mentioned. Try your LOOK command for each item. If there's a DEAD DOORNAIL up there, you want to make sure your program responds when you say LOOK DOORNAIL. If this gets you an I DON'T KNOW WHAT YOU MEAN BY DOORNAIL response, you've got to fix it. That's why we left increments of 20 in the 3000 conversation routine, to fit in all these sorts of additions that are sure to crop up. GOs should work the same way; if there's a CHAIR, the command GO CHAIR should get a meaningful response.

Next, try to take the movable objects. Pick up each one and see what happens and put it down again, performing all the variations on this theme that you can think of. If these don't work, fix them. Again, mark all the changes on the printout as well as making the changes in the program.

The final things to do are everything else. Start at the one-letter-word commands and run up through the nine-letter ones, trying both the ones you've included as well as ones you haven't. Make sure the responses are appropriate for each situation and change them if they're not. Work on both your nouns and your verbs, because there could be all sorts of bugs just

waiting to jump out at you. And when you've finished all that, go into the next room and do it again.

As you progress in this fashion, you'll find fewer and fewer bugs, but there will probably be one or two rough edges that can be ironed out in each room. But eventually you'll catch them all and have the semblance of a finished game. When you think you've got them all, run through the game doing all the right things, solving all your complications, and see what happens. You should win in about five minutes. If you can't win your own game, how is anybody else going to do it? When you're satisfied that everything is working properly, you might want to renumber it all for the sake of neatness and print another listing of the (almost) finished program for your records. Of course, these steps are entirely optional.

The next step in the procedure is highly recommended, but it's a two-edged sword that requires caution. Give the program to an adventure-playing friend and ask him to play it on his own for a while (not under your prying, watchful, snoopy eyes). Tell him to call you if he gets any breaks in the program (he shouldn't get them at this point, but he probably will) or anything else that doesn't gel. But don't have him call you for help in playing the game; that's his problem. The nice thing about having somebody else help with the debugging is that he can find things that you, as the designer with complete knowledge of the inner tickings of the program, might easily overlook. The bad thing, and what makes it a double-edged sword, is that this process is not unlike giving a friend a copy of your great American novel. If your game's a complete stinker, do you really want your friend to tell you that? Will you really accept his opinion that it's too hard or too easy? Will he have good ideas to improve the play of the game, better complications, whatever? And if so, how will you respond? I can't help you out on this, because I have the same problem myself. Designing a game—or any program—is a creative process in which in one respect or another you put your ego on the line for others to tear apart as they see fit. Even abject flattery from your players will do nothing to relieve the fact that you will probably never be completely satisfied with your game. In that regard there is no difference between creating a game or writing a story or making a film or any other imaginative activity. All I can do is quote two truisms. If your work does not succeed, find out where the problem is and work at getting it better. And if your work does seem to succeed, keep at it and improve it until it succeeds even better.

# 6

# TIDBITS AND TRIVIA

In "Space Derelict!" you've seen a lot of programming doing a lot of things and gotten a complete picture of one way of creating computer adventure games. I do want to reiterate a couple of points that are important to keep in mind, especially if you are a relative newcomer to programming. The first of these is that what you have just seen is not the only way of doing it. For my money it's a good, simple way, but it's certainly not the only way. Even as we went through the program I pointed out areas where a tightening of the program statements could make for more elegance, or where perhaps a section could be relocated to enhance processing speed a bit. It is important to remember that this adventure was designed as an instructional tool; if one were so inclined, one could take the exact same game and design it in such a way that it would run identically but so that nobody but the programmer could understand it—which would have had you demanding your money back on this book if I had done so myself. Following to the letter everything I've said here will give you perfectly good adventures; improving on what I've said will give you perfectly better adventures. You have the map now; where you travel is up to you.

Another point that must be made is that this program design is not particularly orthodox in its methodology. The nice thing is, it doesn't have to be, but if you stop reading this book now I don't want you to think that every program can be designed the way this one has been. There is a concept kicked around in computer circles called "structured programming," which has nothing to do with anything we've been discussing so far. In essence, structured programming is a step-by-step process of design where a program is constructed as a series of independent blocks, each block consisting of a particular subroutine to do one small part of the program's whole. For the most part, that is the way programs ought to be designed, and in fact that is

the way our poker game is laid out, so we'll be going into programming theory in some detail in Part Two of this book. But by pointing this out now I feel I've done my duty and silenced in advance those structuralists who will decry me on the basis of just this one program. My credo, which I've made no bones about all along, is that if it works, it works. Although there are some definite rights and wrongs that must be addressed in programming, there is no point in getting hung up in certain academic stylistics just for the sake of those stylistics. If I ever write my book *Zen and the Art of Personal Computing* I'll go into this in more detail, but for now let's resume our regularly scheduled program, which is already in progress. The contents of this chapter are some odds and ends I didn't work into "Space Derelict!" They're not drawn out in glorious Technicolor, full-screen detail, but they'll get you started on just about any other programming situation you might encounter in an adventure.

## AN ELEMENTARY APPLE CLOCK

Some brands of computers come equipped with various sorts of built-in timers, and I've seen ads here and there for clock peripherals you can plug into your Apple. There's not much point to these things for most hobbyists, and they are no help to us as adventure designers.

It is not inconceivable that you might want to incorporate a time factor into an adventure. A bomb might be scheduled to explode at midnight, perhaps a treasure must be discovered before the moon is in its fourth phase—there are all sorts of possibilities along these lines. The problem then becomes figuring out some way to measure the imaginary time taken by the player in each of his moves. The simple part of this is the establishment of the deadline—such and such an action must take place by such and such a time, or else. The hard part is coming up with the measurements.

The most elementary and probably the least satisfying approach to artificial timing is to limit the number of player moves. You might throw in a new variable, PM (for Player Move), and augment this variable every time the player comes around to the beginning of the 3000 conversation segment:

```
3005 PM = PM + 1: IF PM = 100 THEN GOTO losing scenario
```

There's not much subtlety to this approach. It simply hits up PM on every run-through until it equals 100 (or whatever number you decide), and then the program branches off to the losing routine (which should contain a special proviso to advise the player that the reason he has lost is that he spent too much time). This can be augmented a bit by a warning to the player:

```
3005 PM = PM + 1: IF PM > 90 THEN PRINT "YOU ONLY HAVE ";
100 - PM;" MOVES LEFT.": IF PM = 100 THEN losing scenario
```

This at least gives the player some knowledge of where he stands and why.

A nicer way to handle this, which is not all that much more complicated, is to give a starting time and a finishing time and break down the difference into recognizable segments. Let's say your scenario starts at noon and the deadline is midnight. That gives the player twelve hours. Let's divide each hour into 12 moves, which means that the player has 144 moves in which to

win the game. After every 12 moves you could tell the player that another hour has just passed and give him a new reading on the time:

```
3005 PM = PM + 1: IF PM/12 = INT (PM/12) THEN HR = HR + 1: PRINT "IT IS NOW ";HR;" O'CLOCK.": PRINT "YOU HAVE ";12 - HR;" HOURS LEFT.": IF HR = 12 THEN GOTO losing scenario
```

This is how we would advise the player that an hour has passed. Each time we come to the beginning of the 3000 segment we again augment PM. The second statement on the line (IF PM/12, etc.) handles the hourly divisions. INT(PM/12) is, of course, a whole number, an integer—that is the definition of the INT command. What we're saying here is that if PM divided by 12 equals a whole number, then this means that one hour has passed, and the HR (hour, of course) variable is augmented accordingly. When PM reaches 144, HR will equal 12, which means that the player will see the line:

```
YOU HAVE 0 HOURS LEFT
```

and will immediately find himself facing your losing scenario.

This can, if you are so inclined, be stretched into days. Let's say we're starting at midnight and we want to give the player the next three days. In that case we would simply plug in:

```
IF HR = 24 THEN DA = DA + 1: HR = 0: IF DA = 4 THEN losing routine
```

This follows the same logic as before, merely extending it by dragging in a new variable. A further elaboration could allow the player to input the command TIME or TELL TIME. We would respond in the conversation segment:

```
IF C$ = "TELL TIME" THEN PRINT "IT IS DAY NUMBER ";DA: PRINT "THE TIME IS NOW ";HR; "O'CLOCK."
```

You could make this even cuter if you wanted by including PM as well and giving hours and minutes. We're starting to get lengthy here, so I'll let you contend with that bit of business yourself.

The next step in making this even more sophisticated is not to penalize the player for moves that don't actually happen. Let's say the player enters the garbage command TAKE CRONW instead of the correct TAKE CROWN. Should he have to pay the price for this? I say no. But this requires a bit of work. It means you have to go through the conversation segment catching each and every instance where a player input is kicked back rather than acted on. Simply adding the line GOSUB 49000 right before the GOTO 3000 in these cowcatcher lines will send you off to:

```
49000 PM = PM - 1: RETURN
```

With this step the incrementation of PM that occurs in 3005 will actually be resetting PM back to what it was before the last bad move was made.

Finally, there is the bend-over-backward version of all of this recommended only for the most masochistic programmers. When you think about it, every action taken by the player does not in fact require the same amount of imaginary time. Going north through an open door is a much quicker process than defusing a 50-megaton nuclear bomb, for instance. Swimming north through a swamp takes longer than running north on a road, and so

forth. Most of your player activities will be roughly similar in time elapsed, but some are different from others and can be handled with a nod to these differences. Are you ready to go through your conversation segment and augment PM by different values for every different activity, with a default value of 1 to be handled as before at 3005? What a nightmare—but it could be done if you were willing.

As with anything else in your adventures, if you're going to put in a clock, put it to good use. Simply limiting player moves has its point, but as long as you're ticking away the hours, why not make the most of it? You could have a clock go off (the appropriate number of Control Gs) whenever the hour is struck. You could have ghosts pop out only at midnight. Or werewolves pop out at the hour of the wolf, 4:00 AM. You could have lunch appear on the table at noon, dinner at eight, tea service at four. As with everything else in adventures, the possibilities are endless if this particular sort of complication tickles your fancy.

## VERIFYING THE SECOND WORD

It is possible to include in your adventure program a definitive routine that would analyze the second word of the player input and verify whether that word was one your program recognizes. This is not very hard to do, but it does require that you comb through your game to pull out every possible second word so the analysis can be made. If you miss even one word, the whole thing is for naught. So it's time-consuming and nit-picky and very much optional (which is why I omitted it in "Space Derelict!"), but it is a nice piece of polish you might want to incorporate.

To pull off this hat trick we're going to use the good old READ . . . DATA statements. If you know what these are all about you can skip down a few paragraphs, but if you're like me you probably never found much use for these oddities (or maybe you don't understand them at all), so bear with me for a while.

The easiest way to describe these two babies is to compare them to the sequential text files in which we stored our saved games. Like a text file, DATA is a storage container, only within a program rather than onto a disk. And READ performs exactly the same function as a program statement as it does as a DOS command: It reads through the DATA (or, in the case of a file, it reads through the fields) from start to finish (unless some other modifying parameter is set), doing nothing whatever other than making the data live within your program for other statements in the program to operate on. The problem is, it's very hard to see much purpose to this operation in the abstract.

Here's an example of READ . . . DATA:

```
10 FOR X = 1 TO 5: READ D$: PRINT D$: NEXT
1034 DATA 1,97,43,16,HIKE
```

You're probably somewhat aware of the basic working of READ . . . DATA statements, but we'll restate it here. Whenever a READ statement is encountered in a program, the computer runs down to DATA and READs off a piece of it. The first READ reads the first piece, the second READ the second

piece, and so on. If there's so much data that you need more than one DATA statement, READ simply proceeds from the first to the second to the third, in order, reading each piece as required. So in the example above, running the program would give us the following results on our monitor:

```
1
97
43
16
HIKE
```

A change of line 1034 to:

```
1034 DATA 1,97,43
1035 DATA 16, HIKE
```

would give us the exact same results. Since we are calling these pieces up as strings (D$), no quotes are used and the numbers are treated as a string just as the HIKE is. If we had used all numbers, a simple READ D would have sufficed.

In the example given, when we get to the end of line 10 (that is, when we hit the fifth NEXT), we have exhausted the DATA storage. If we're finished with it, that's fine, but if we need it again, we could add:

```
20 RESTORE
```

so that the pointer that was making sure we took each piece in order will now be set back at the first piece, so that a subsequent READ statement would start at the beginning again (with the number 1 in line 1034).

Of course, we don't always have to READ all the pieces in a DATA statement if we don't want to.

```
10 PRINT "PICK A NUMBER FROM 0 TO 9"
20 GET X
30 READ Z
40 IF X < > Z THEN PRINT "YOU LOSE!": GOTO 10
50 PRINT "YOU WIN!": RESTORE: GOTO 10
60 DATA 3,0,9,4,1,7,2,2,4
```

Here's an extra game thrown in at no extra cost, and if it's dull as dishwater, at least it demonstrates READ . . . DATA. The way this game will proceed is simple. The player will be prompted for a number from 0 to 9. After GETting the number the program READs the first DATA element, in this case the number 3. If the player has guessed incorrectly, we go back to line 10 for another try. This time the correct answer is 0. Wrong again? Back to 10, the correct answer is 9, and so forth. But if the player guesses correctly at any point, then the RESTORE statement in line 50 puts the pointer back at the first element, the number 3, and the whole thing starts over again.

Unfortunately, there is a flaw in this extra give-away-for-nothing game. What happens if a player makes nine incorrect guesses? At the tenth run-through there are no more pieces of data to be read, so I'm afraid the program will crash as a result. But there's a way around this, too.

```
5 IF Z = 10 THEN RESTORE
10 PRINT "PICK A NUMBER FROM 0 TO 9"
20 GET X
30 READ Z
40 IF X < > Z THEN PRINT "YOU LOSE!"
50 IF X = Z THEN PRINT "YOU WIN!": RESTORE
55 GOTO 5
60 DATA 3,0,9,4,1,7,2,2,4,10
```

Now we've covered our flanks 100 percent even though we've thrown in a ringer. The GET statement limits our player to one digit input, of course, and therefore there's no way for him to input the number 10, so that extra number at the end of our DATA statement is a signal to the program that the DATA have run out; line 5 will handle that situation for us by making sure we never run out of pieces.

However, none of this really gives you a good idea of what else READ . . . DATA statements are used for. The most common use I've seen is for large-scale POKEing of numbers into a program either for sound-effect generation or graphics. If you are using a shape table in a Basic program, this means you will have to load in a lot of very special bytes in very special places. You might have 20 shapes, each one requiring a series of 10 or 20 numbers, meaning you might easily have a couple of hundred elements where each number has to be POKEd into specific memory addresses individually. In these cases you'll see something like this:

```
10 FOR X = 768 TO 900
20 READ Z: POKE X,Z
30 NEXT
40 DATA    233 numbers one right after the other in a series that makes no
```

sense whatever except that they probably draw nice little machine guns and aliens all over your video monitor.

But we're not going to go into graphics here yet, so where does all this leave us adventure designers?

There is a very easy way of taking the player input right off the top and making sure that the noun—the second word of the input—is one we have covered in the conversation segment. The first thing we have to do is break up C$ into the two elements S$ and S2$. If we have no S2$ (i.e., C$ is a one-word input), everything would continue as normal. If we do have an S2$ we would then branch off to somewhere like line 49000 and try this:

```
49000 REM SECOND WORD CHECKING SUBROUTINE
49010 READ SW$
49020 IF SW$ = "FINIS" THEN PRINT "I DON'T KNOW WHAT" +
S2$ + "IS": RESTORE: RETURN
49030 IF MID$(52$,2) = SW$ THEN RESTORE: RETURN
49040 GOTO 49010
49050 DATA DOOR, HOLOGRAM, NORTH, SOUTH, PANEL, CEIL-
ING, DEVICE, et cetera, et cetera, et cetera,FINIS
```

This would do the job very neatly. If we get a match in line 49030, then the

subroutine is terminated, the DATA restored, and the chips will fall where they may in the remainder of the conversation with the computer segment. If we get to the end, which is FINIS—the last piece of data—then we print the message that the word is a dud and go back whence we came. Keep in mind that a cowcatcher is now necessary immediately following the line that got us here in the first place (the GOSUB 49000) to the effect IF SW$ = "FINIS" THEN 3000, which would get us back to the beginning of the conversation segment. Deleting the RETURN and substituting a POP in line 49020 followed by GOTO 3000 would have the same effect—but with all this READ . . . DATA information flying around at this point we're not really ready to explain POP yet. Besides, I'll need *something* to talk about later in this chapter.

## CHANGING A GAME IN PROGRESS

Everything we've seen so far has given us innumerable possibilities for game design, but there has been one limitation that we only broached briefly in our discussion of miscellaneous business. Up till now the actual adventure has been static; although the player moves around in it however he may choose, at his own pace and his own inclination, the game itself remains the same. Unless he takes movable object X from room Y, movable object X will always remain in room Y. The game is entirely at the command of the player, entirely lacking a kinetic energy of its own. The player just goes along and the game universe just sits there as he roams around in it. There is nothing wrong with this per se, and some very interesting and classic adventures have had this quality about them. But there is no reason why we can't expand our game universe and give it a bit of life on its own, and that's what we'll be talking about now.

Right off the bat, I don't want you to think I am proposing the introduction of random factors into an adventure. Quite the contrary: As far as I'm concerned, anything random in an adventure goes counter to the whole point of what an adventure is all about (although one might make an exception for some minor randomizations—the ability of a player's gun to hit a specific target, that sort of thing). An adventure should be a well-thought-out puzzle with dozens of twists and turns to challenge the player's intellect. Once things start becoming random the challenge is less on the player's intellect and more on his hunches or his reflexes—fine for an arcade game but not the stuff that adventures are made of.

The kinds of changes I have in mind here are in the nature of the normal comings and goings of people and things that would normally transpire in a real universe similar to the one in your game. In the basic module they belong in the line 40000 miscellaneous-business area. You can make these incredibly complex or quite simple, depending on how deeply you want to dive into it. Say, for instance, that you have a snake in your game, out to attack our player. Of course, you could just keep the little viper waiting in a given room, ready to pounce, but being an amateur herpetologist you feel your snake should laze in the sunshine during the day and laze in his den during the night. No problem. You build in a clock as we discussed before and decide which hours you want to be sunlight and which darkness. Then

you change the CO(X) value of your snake to reflect the appropriate sunny or dark room wherein the snake is lurking. This is a simple enough process and is easily adaptable to a number of circumstances.

The example above requires a clock, but you might want a different sort of action dependent not on independent factors but on the player's moves. For instance, you might have in your game a vampire who is stalking the player. The vampire could move in a random pattern from room to room (this is another example of reasonable randomness), but wouldn't it be much more clever to have him able to sense where the player is in order to track him down? Again, no problem, but this time a lot of work. The requirement here is a room pattern laid out so that whenever your program knows both the R value of the player and the CO(X) of the vampire, the program is able to decide which adjacent room to move the vampire into in order to put it that much closer to the player. No mean feat, that. It requires one of two things: a room layout such that a simple mathematical formula will always quantify the differences between two rooms and provide the next closest room from one to the other, or else a very dull programming operation where every room is analyzed in relation to every other room, giving the closest in each case. The first option is infinitely more desirable but requires much work at the conceptual stage. The second option simply requires a lot of dull computer hacking.

There are other ways to move things around, however. You could base a movement on the fulfillment by the player of some special criterion. For example, let's say that if the player enters the restaurant through the front doorway, the sinister dishwasher exits through the back one. This is simple enough to handle in the GOs. Or you could give a game character an entire itinerary to follow and simply have that character move in a prescribed fashion from room to room with the completion of each player input. Again, this would fall into the line 40000 miscellaneous-business section and would require a complex loop monitored by a special variable:

```
41320 REM KGB PETE MOVEMENTS
41330 KG = KG + 1
41340 IF KG > whatever the limit is to your loop THEN KG = 0:
GOTO 41330: REM START LOOP AGAIN
41350 ON KG GOTO a number of different places immediately
described below starting with 41360
41360 CO(99) = 101: GOTO exit from routine: REM CO$(99) = "KGB
PETE"
41370 CO(99) = 102: GOTO exit
41380 and so forth and so on until
41490 REM EXIT FROM KGB PETE ROUTINE
41500 IF CO(99) < > R+100 THEN RETURN
41510 POP: GOTO 50000: REM DISPLAY PETE ON SCREEN
```

There's that damned POP again. This time I guess I'm going to have to explain it.

These lines above are fairly straightforward and understandable with all we've covered so far, until you get near the end. Line 41500 covers the condition that KGB Pete and your player are not in the same room. If that is

the case, the room display at the top of the monitor need not reflect Pete's presence, so you can return from the subroutine without a care. But if Pete *is* in the same room, you want to make his presence known at the top of your videoscreen, which means making a trip to the 50000 room-layout routine. The problem is, we are now in the middle of a GOSUB, with the computer waiting for a RETURN, whereas in fact we want to bust out of this subroutine and continue processing somewhere else entirely. That is where POP comes in. POP is a useful fellow who doesn't make much sense when you first run into him in the manuals, but once you see him in action you can begin to appreciate him. You see, whenever you encounter a GOSUB statement in a program, the computer puts a tick in its memory so it will know where to come back to when a RETURN is encountered. You can have GOSUBs within GOSUBs, which, of course, is called "nesting," which means that you will have multiple ticks in the memory, and the computer will always refer to the most recent one each time it encounters a RETURN. The problem we face now is that we don't *want* to RETURN. We want to go to 50000 and get out of this GOSUB predicament altogether. POP will do that for us by erasing the most recent GOSUB tick in the computer's memory, allowing processing to continue as if that last GOSUB had never existed (makes you wish you had a FOR . . . NEXT POPper as well, doesn't it?). So the POP in line 41500 allows us to go wherever we want without the penalty of an unreturned RETURN hanging over our heads. Without this, the error here would show up way down the line and with much subtlety (a crash with an OUT OF MEMORY ERROR or the like), which is very hard to handle when it comes to debugging. Just as there are no RETURNs without GOSUBs, there can be no GOSUBs without RETURNs unless we POP that old weasel before it gets out of hand.

Anyhow, we have covered in this section a few possibilities for internal movement within your universe. As you can see, none of these things is a simple one-liner. Although for the most part the programming is relatively easy, it can be long and time-consuming. So use these ideas wisely and frugally. They can liven up a game, but they can also deaden up a game designer.

## NOTES ON A D&D SCENARIO

I would find it very surprising if anyone reading this book didn't know what D&D—"Dungeons and Dragons"—is all about. So many adventure games are direct derivations of the D&D phenomenon that I simply can't imagine a computer hacker being unaware of so seminal an object. Only a few years ago you had to seek out D&D literature far and wide; now it's available in almost any bookstore. But for those who have never played, one brief paragraph before we get down to business.

Like a computer adventure, "Dungeons and Dragons" (and its various offshoots—the original is very much protected by trademark by a company called TSR Hobbies, Inc.) is a role-playing game. An artificial universe is created for the players to roam around in, on quests either for treasure or to perform specific tasks. Players take on the characteristics of warriors or wizards, thieves or clerics, whatever suits their fancy, human or otherwise.

As time passes, a player character gains experience, and with that experience comes increased power and knowledge. The rules for D&D are not etched in stone (although there is much guideline literature), nor is the game universe a fixed, given commodity (although some prefabricated universes are available in the stores). The structure of the game rests with the player chosen as Dungeon Master. This player designs the universe, peoples it with nonplayer characters, stocks it with treasures, spices it up with traps. This player moderates the moves of the player characters, arbitrates what can and cannot happen within that universe, and consults the manuals when the manuals need consulting. Not unlike an adventure game designer, eh?

The big problem with getting a good D&D game going is that you need a fairly large group of dedicated players. A single game can take months, while a player character can last through dozens of games. A band of dedicated comrades-in-arms is hard to find outside of school, the army, or very weird job environments, and those are where most successful games are conducted. But for solo operators, the computer provides a close second, via the adventure game.

The role-playing in a computer game is intrinsically related to the role-playing in D&D, but there are some elements of this sort of game that we have not covered yet and that we should touch upon briefly. The biggest difference is the element of combat, which we've only barely handled in our attack of the killer robot in room 23 of "Space Derelict!" But let's say you want dozens of killer thises and thats roaming your hallways, and more, you want your player to be able to kill them not only by wits but also by brute strength, if such strength is called for in your scenario. This requires that both your player and your killers be imbued with quantifiable attributes that can be measured against each other, with a slight random agent thrown in for good measure. Let's say you boil it down to three variables: ST for strength, SP for speed and reflexes, and WI for wiliness and wit. An average man would have average strength, average speed, a bit better than average wiliness. A superman type would have the same wiliness, most likely, but greatly increased strength and speed. A dwarf would have greater-than-average strength, less-than-average speed, and the same wiliness as the average man. A wizard would have less speed and strength than the average man but much greater wit. A tiger would have strength and speed and no wit, a snake speed and wit but no strength, a furgle-eyed difflesnopper would have whatever you'd want to give it.

Each combatant would have a number applied to each of his three variables—from 1 to 20, say—and this would begin to account for his possibilities in combat against another combatant. It is up to you as designer to decide just what beats what in the actual arena. Will an SP of 9, an ST of 15, and a WI of 4 beat an SP of 2, an ST of 6, and a WI of 19? Who knows? You must include in your game a formula for determining just what beats what. Such a formula is known in programming as an algorithm. Algorithms make the computer world go round. The algorithms that guide the activity of a computer in a given situation—where certain seemingly unquantifiable concepts are indeed quantified, analyzed, and acted upon—are the real crux of what program design is all about, and we'll go into them further when we talk about designing strategy games. For now I'll suggest only how the SP, ST, and WI algorithms might work within an actual program. A player would

encounter an enemy and first be given the option to run or fight. If he runs, the enemy gets a free shot at his back, but with only slim chances of getting a hit. If a hit is made, then the player must turn and face combat; if he's not hit, he's home free for the time being. If combat is engaged, you either have to set it up as a series of hits and measure the success or failure of each hit with your combat algorithm, or else your algorithm will simply say IT'S DEAD or YOU'RE DEAD. The former approach has a bit more style to it.

None of these combat variables should remain static. I would imagine one's strength, for instance, would be the measure of one's existence. If your enemy keeps hitting you, you lose more strength until your strength is zero, and it seems to me that that would make you a goner. But a 97-pound-weakling character ought to be able to gird his loins and get himself some weaponry to up his strength a bit. A player with a 10 in ST could have that increased to 15, say, if he's got a mace tied to his armor (or a can of Mace in his pocket—who knows what kind of game you're designing here? For the record, I use the masculine pronouns throughout this book both for the sake of convenience and because, let's face it, this kind of game-playing does tend to be more a male than a female diversion). Of course, once you start getting into weaponry you most likely have to create a whole arsenal, with each individual item not only accounted for as a CO(X), but also with its ST thrown in for good measure. Then there's the question of magic weapons that have special powers—perhaps secret, perhaps not. There are magic spells available that would certainly up anybody on the WI scale. When you start getting into this you can begin to see why a good D&D-type game is hard to find. Not only do you have all the elements of your basic adventure, but also a whole new batch ultimately just as complicated thrown in on top of them. My suggestion would be to ease into this sort of thing slowly after you're fairly confident about manipulating the basic adventure module. Don't start by throwing in 100 different monsters of all shapes and varieties; two or three will be good enough, with a simple, boiled-down algorithm to guide them, and as you watch them shake down in actual play you'll begin to get a feel for how they could be improved and how your algorithm could be expanded to make the combat feature that much better.

There is another idea I've had on D&D for a while, and I even know one person who tried it but ultimately abandoned the idea when he realized just how much chewing he had facing him after he took that first bite. Being a Dungeon Master is not easy and requires a lot of looking up data, throwing dice, that sort of thing. Why not write a program for Dungeon Masters that does all of the dogbody work for them? They'd still have to arbitrate the squeakers and orchestrate the universe, but they wouldn't constantly be figuring out this one's hit points vs. that one's strength, with the plus-two sword factor and all that sort of stuff. I wouldn't be surprised if such an item is already on the market, although I've never seen it, but if you're an experienced enough Dungeon Master you wouldn't need much programming savvy for what would basically be a simple sorting-and-storage program.

## KEEPING PLAYERS OUT!

Here's the problem. You've written this wonderful adventure that's taken you months and months to perfect, and some clown comes along and breaks

right into it and starts taking the program apart bit by bit to find the solutions not by playing it but unscrewing it. Unfortunately, there isn't much you can do that's really effective to protect your average, everyday program. The sort of copy-protection that accompanies the software you buy in the stores is certainly economically unfeasible to amateurs (and, quite frankly, akin to shooting gnats with bazookas when applied to homegrown products). And let's face it, if the determined software pirate can break into disks that the pros have protected, there's not much you can do against the Sunday-morning pirate who wants to break into yours. Nonetheless, there are various tricks you can employ that will prove moderately frustrating, although they're not much more than temporary stopgaps.

The first thing you want to do—and this is where you will have your biggest problem—is keep your opponent from simply putting your disk into the disk drive, typing CATALOG, finding out the name under which you saved your game, then typing LOAD GAMENAME and finally LIST GAMENAME. Second, you want to be able to keep your pirate from pressing CONTROL C in the middle of a game, thereby breaking in that way. The following is a weak but worth-a-try form of break-and-entry protection that will work against a less-than-determined burglar:

```
1 POKE 214,255
2 ONERR GOTO 45000 (or other similar open line)
45000 IF PEEK(222) = 255 THEN RESUME
```

On top of this, initialize your game as the HELLO program on the disk. That ought to hold them for a little while.

What have we done here? First, by actually initializing the disk with our game, we have constructed a situation where the average player will feel beholden to run your program by booting DOS (I'm assuming you've already read the first few pages of your manuals so you know what I mean by booting DOS—essentially this is pressing the "on" button). This means that when the program runs—and now we've set it up so the player feels he must boot to run the program because it is the HELLO (which is not true, of course, because HELLO programs can be LOADed or RUN just like any other)—he will first hit those few lines of programming above. And these few lines do some interesting things. Line 1 turns just about everything into a RUN command. So if this line itself gets run, until this 255 value is POKEd out of address 214, commands such as LIST will RUN the program. To compound this further, we have our ONERR routine. The ONERR command is just what is sounds like; it tells the computer that if there is an error, do the following. It doesn't matter what the error is that the computer encounters after reading an ONERR, and it doesn't matter on what line in the program the error occurs. Unless that original ONERR is countermanded somewhere else in the flow of the program, the computer will continue to remember your ONERR instruction, and act accordingly when the time comes. In this case we branch to line 45000.

The purpose of ONERR is to allow the computer to continue processing despite errors, and it's not unusual to see an error-handling routine tagged on at the end of fairly complex programs, especially when those programs are being run by your so-called users and not the programmers themselves. The purpose of these routines is not to detect errors in undebugged programs—Applesoft is perfectly capable of doing that for you for the most

part—but to keep a program up without losing data if the user does something incredibly stupid (which has been known to happen). The ONERR routine will usually print to the screen something like "ERROR—PLEASE CONTACT YOUR PROGRAMMER" followed by detailed analysis of just what went wrong for the programmer to fix. Never, never, NEVER put an ONERR routine into a program you are not sure is 100 percent debugged, or you'll either drive yourself batty or else simply be overriding the whole point of the intrinsic Applesoft error-handling routines.

In any case, back to our antiburglar routine. At line 45000 we see a PEEK into machine address 222. This address is the home of the number that tells the computer what error has just been made so that the computer can tell you. Let's say you have a mathematical formula in your program where at some point a number is unexpectedly divided by zero. This is unacceptable, of course, so your program will break with an error. The value of address 222 will become 133, which is the numeric token for the division-by-zero error. If the value of address 222 is 0, in another f'rinstance, then we're talking NEXT without FOR error, and so forth (there's a complete listing of these in your manuals). In our routine we're looking for 222 to have a value of 255, which is a Control C interrupt of the program (which the computer treats as if it were an error). Try it. Break into any program with a Control C and then type:

```
PRINT PEEK (222)
```

If you don't get an answer of 255, drop me a line and I'll eat my hat. What we're doing in line 45000 is this: If the player tries to sneak in a CTRL-C break, no matter where he tries it, the computer will remember the ONERR instruction, will branch to 45000, and the player is condemned to go back to where he left off in the program, courtesy of the RESUME command, which does exactly what it sounds like it does. In other words, Gotcha! So combining this error-handling routine with our RUN command-override, we've now got our pirate into a bit of a bind. He can get out of it with a little thought, but is it really worth it for him to try? Probably not, and it's only cost us three lines of programming, so what the hell. But remember, don't put that ONERR routine into an undebugged program, or else every time you hit even the most miserly syntax error you'll branch to 45000, and Lord knows what will happen after that.

## GETTING MORE MEMORY FOR YOUR MONEY

As I've said a number of times, the adventure module we've developed here is only one way of doing it. The fact that it's one of the easiest ways was our reason for using it. I have seen other much more sophisticated styles of Basic adventure design, some of which are as complicated to write as they are to solve. Nonetheless, these more sophisticated designs usually serve one very important purpose we have ignored entirely: They take up less memory in the computer. But 64K seems like more than enough, you say? Not exactly, because as an Applesoft programmer you have nowhere near the advertised 64K to work with. On a II+ with 64K, for instance, you have roughly 35,000 bytes available; the other 30,000 or so are used for the computer's operation. If you're a good fudger you might be able to expand your Basic

space a little by relocating some of the internal systems business, but this is pretty far-out stuff requiring quite a bit of knowledge of machine language, DOS, and so forth. Where does that leave us? With only about half the memory we thought we had. Don't feel too bad, though. Your share of the Apple 64K is a lot larger than the programming shares of 64K given to most other machines on the market. Every cloud has its silver lining.

If you were to enter "Space Derelict!" exactly as is into your machine, you would eat away just about all the memory space available to you. Any other adventure using this module will do roughly the same thing. It is not inconceivable that you could run out of available memory with your own game. If you trim down the module using some of the structuring techniques discussed in Part Two of this book you might avoid the problem, but there's another solution as well. You might like this solution well enough to use it all the time. The look of the game will change quite a bit, but it still will be the same game. You can store all the room information in separate sequential disk files, thus freeing up most of the program space from line 50000 up. Instead of using the text window with the steady room display, you would scroll the room information, displaying it on the screen whenever the player enters a new room or specifically requests a room description. And you'd get a lot of extra memory for the rest of the module.

To do this, first we change the line 50000 subroutine to look like this (bear with me—this is pretty hard-core stuff):

```
50000 REM ROOMS
50010 RFILE$ = "R" + STR$(R)
50020 PRINT CHR$(4); "OPEN "+ RFILE$
50030 PRINT CHR$(4); "READ "+ RFILE$
50040 FOR X = 1 TO 3
50050 INPUT RO$(X)
50060 NEXT
50070 INPUT N
50080 INPUT S
50090 INPUT E
50100 INPUT W
50110 PRINT CHR$(4); "CLOSE "+ RFILE$
50120 FOR X = 1 TO 3
50130 IF LEN (RO$(X)) > 0 THEN PRINT RO$(X)
50140 NEXT
50150 GOSUB 13000: PRINT R$: REM ITEMS IN ROOM
50160 IF N=1 THEN PRINT "I CAN GO NORTH.      ";
50170 IF S=1 THEN PRINT "I CAN GO SOUTH.      ";
50180 IF E=1 THEN PRINT "I CAN GO EAST.       ";
50190 IF W=1 THEN PRINT "I CAN GO WEST.       "
50200 GOTO 3000
```

Here we're opening a file called RFILE$, where RFILE$ equals the letter R plus the number of the room. Then we're printing out the information already contained in the RO$ strings, followed by a trip to 13000 to see what movable objects are lying about, followed by direction information based on the values of N, S, E, and W, followed finally by a trip back to the beginning of the line 3000 conversation with the computer. Now whenever we make a

move from one room to another we will come to line 50000, and instead of having an ON statement leading us to the correct room description for immediate printing, we will open a specific file, read out the data, and then go back to our conversation. No matter which room we're in, it always will work the same way.

The next thing we have to do is create all the RFILE$s, with their RO$s and directional data. We would create a new program called RMAKER:

```
10 INPUT "NUMBER OF ROOM";R
20 RFILE$ = "R" + STR$(R)
30 FOR X = 1 TO 3
40 INPUT RO$(X)
50 NEXT
60 INPUT "N";N
70 INPUT "S";S
80 INPUT "E";E
90 INPUT "W";W
100 PRINT CHR$(4);"OPEN " + RFILE$
110 PRINT CHR$(4);"WRITE " + RFILE$
120 FOR X = 1 TO 3
130 PRINT RO$(X)
140 NEXT
150 PRINT N
160 PRINT S
170 PRINT E
180 PRINT W
190 PRINT CHR$(4);"CLOSE " + RFILE$
200 GOTO 10
```

When you RUN this program you first will be prompted for the room number. Then you will be prompted for three lines of text, RO$(1) through RO$(3). You would now enter your room descriptions. Let's say we're using this technique for "Space Derelict!" Starting with room 1 we would INPUT "1" followed by a line of text I AM HANGING ON TO A HAND HOLD AT THE ONLY ENTRANCE TO THE SHIP. This would become RO$(1). Then we would INPUT RO$(2), AN OPEN AIRLOCK LEADS BEYOND MY LINE OF SIGHT. We would have no RO$(3), so we'd simply press Return, making RO$(3) a null string. Then as we'd also have no N, S, E, or W in room 1, we would again press Return when prompted for each of these, thus giving each a default value of 0. Then the program would create the file R1 and write it to the disk, then loop around to do it again. This time we would enter room 2, and so forth. Whenever we didn't have data, we would just press Return. If we had exits in a given direction, we would INPUT a 1 for that direction. In room 3, for instance, which has exits south, east, and west, we would have no INPUT for N, followed by an INPUT of 1 for S, 1 for E, and 1 for W. The routine in the game module at 50000 would know whether to print I CAN GO WHEREVER, depending on the value of each of the directional variables. Since every time we access 50000 we also access the disk file, we don't have to worry about deleting values of these variables from room to room, as the loading in of new values (even if those values are 0) will do it for us (be careful not to use N as a counter variable in other subroutines, though, or you could screw up the north variable).

Using this technique requires one other major change in the program. Since this room description information will be scrolling normally off the screen whenever something new is printed below it, it is not unlikely that sooner or later our player will be in a room without any room description in front of him and will want to see such a description. For that eventuality we amend the LOOK section of the conversation with the computer so the C$ of LOOK or LOOK ROOM (or the synonyms DESCRIBE and EXAMINE) will branch to 50120, where we begin printing to the screen the various RO$s, (if we haven't changed rooms there's no need to reaccess the disk to get the same information already contained in these variables). Note also that in 50130 we print RO$ only if it is not a null string: This allows us to have room descriptions of varying lengths without printing blank lines on the screen.

This technique of storing the rooms in files is *not* recommended unless you're really adept at handling text files. Making changes, for instance, can be touch and go, and what's worse, you can't use commas in your descriptions. (In an INPUT statement Basic interprets a comma as if it were a carriage return, not a character. You can get around this in various ways if you know how, but it's too complicated for us to go into it here, so I advise avoiding commas.) If you're writing adventures that long, I first recommend trimming down what you've got rather than adding an extra possible nightmare. Nonetheless, if you feel up to it, you might find it quite helpful.

Finally, if you're really up against it spacewise, you might consider breaking out complete segments of the total program and creating a small series of shorter programs, linking them with the CHAIN utility on the system disk that came with your computer. This machine-language program, detailed in your DOS manual, allows you to connect programs, carrying the values of the variables from one to the next. It could be especially simple to CHAIN from the introductory material to the conversation with the computer and perhaps out of the conversation to your winning and losing scenarios. The manual explains how to do this, so I won't repeat it here.

## WHY GRAPHICS?

Why graphics indeed! I mentioned at the very beginning that although more and more professional-level adventure software now contains high-resolution graphics, these graphics in themselves don't make or break the game. For the most part these graphics are window dressing; they can be very attractive (although they can also look pretty kindergartenish), but the game could exist perfectly well without them. It's the difference between room descriptions in words, which are capable of containing all the verbal subtlety you as the designer can come up with, vs. pictorial room descriptions, which are limited not only by the programming abilities of the designer but also by the nature of the graphics beast itself. The Apple can do some very nice graphics, thank you, but it is certainly not the most versatile computer in the graphics area. How many people actually own a color monitor, for instance? A lot of people get to look at all those nice color graphics that people worked so hard to develop only on a standard computer monitor, in delicious dark green and light green. Or worse, you're fighting with the rest of the family

over whether the TV gets hooked up to the computer or to reruns of *All in the Family.* A color monitor costs even more than a regular color TV, and most people, after investing in the real computer essentials, tend to draw the line at this final investment. Can't blame 'em, because I originally did it myself.

Anyhow, the problem with hi-res adventures for the most part seems to be that the graphics are more for the sake of commerciality (the state of the market almost demands them) than for the game itself, in which they play no integral part. Some designers have tried to break through that window-dressing barrier, but so far the few results I have seen haven't really been successful (which is not to comment on the designers' successes with the games themselves, just the integration of graphics with the adventure). For instance, I've seen your standard hi-res adventure picture come on the screen (an odd chair, door, window), and all of a sudden an animated mouse passes across the floor. Nice touch, but the original graphics still have that kindergarten woodenness. The mouse is the best part. And I've seen ads for at least one adventure that halts in midgame to integrate an arcade action scenario—very commercial indeed, but too many steps taken away from our goal to succeed in the original problem of graphics integration in adventures.

You, of course, may be a dedicated devotee of graphics adventures and, despite my grumpiness, might be very disappointed that I have chosen to circumvent the subject. Well, there's a reason for that beyond my personal preferences: The subject of graphics is a book in itself. The problem is that unlike the things we've been doing so far—moving variables, displaying text, simple filing, all of which are handled quite excellently by Basic—graphics require one of two things: a slightly more than rudimentary understanding of bits and bytes and the whole hexadecimal universe, or a good purchased graphics program to handle all that stuff for you. If you go the first route and stick with it, you'll probably end up a 6502 machine-language programmer. If you go with a packaged program, you'll probably get quite good results neither knowing nor caring about the whys and wherefores.

The problem remains, having gone one of the above two routes, how do you integrate the pictures you have drawn into the basic module? Simple enough: You just substitute your drawing routine for the 50000 room-layout routine. If a player enters room 1 you BLOAD ROOMONE, then POKE the soft switches to display the room-1 picture, then display a little line of text or two describing the room and its contents. In addition, you integrate an Escape function so that if the player input (C$) equals the CHR$ value of Escape, the picture is either removed by soft switches and the most recent text is displayed, or vice versa. It's as simple as that, and if you don't understand a word of it, I'm sorry but it really would take a whole other book to make it ABC-understandable. You don't have to worry, though, because if you decide you want graphics and buy a packaged utility, the documentation with that package will either explain BLOADing and soft switches or will somehow do it for you.

The point is, after you go through the whole business of creating a series of high-resolution pictures—no mean feat—do they really do anything to improve your adventure? I don't think anyone has yet solved this basic problem. If you think you can solve it, be my guest, but be prepared for countless hours of additional work and for some disappointments along the

way. Art is art no matter how you slice it, and if you can't draw a straight line with a pencil don't expect to be able to do it with a computer. *Get help* if you're not a good artist, because otherwise you'll end up as frustrated as a paternal mule. But if you take my advice, you'll forget about the graphics part of it, at least as far as adventures are concerned. They just don't really do anything.

# PART 2

# STRATEGY GAMES

# 7

# THE ART OF STRATEGY

## GAMES, STRATEGY, AND ASSORTED BALONEY

I debated with myself for quite some time whether to burden this book with all my theories about games and game-playing. There is much to be said on the subject, and much has been said and continues to be said almost daily by various experts and pseudoexperts. The question of what games are and why they are played seems to be the concern of just about every branch of both the physical and the social sciences. Anthropologists wonder about game-playing among chimpanzees, psychologists try to make sense of children playing kick the can, sociologists equate business management with game-playing, and there is even a branch of mathematics devoted to game theory. Admittedly all these people tend to have different purposes in their studies, but nonetheless there is much overlap in what they're studying if not exactly in the way they're studying it. The question then is, why? Why have all these serious Ph.D. sorts taken it to themselves to try to figure out what the apparent frivolity of gaming is all about? In a way it depends on which Ph.D. you happen to be talking about, but one thing is certain: Games are a major facet of human (and perhaps in some cases nonhuman) existence. People play games; apes play games; cats and dogs play games. And it's curious that now that we've created artificial intelligence—computers—one of the first things we've done is teach our computers to play games as well. Score another point for all those Ph.D.s.

As I said, I have a lot of theories about games, but I really don't have all that much to back them up, and this is not the place to go into them anyhow. But now that we are in a position to make game players out of computers, we ought to have some idea of what games are all about so we'll be able to go about our game-designing in the best way possible. And the first thing we

have to do is disabuse ourselves of the idea that adventures such as "Space Derelict!" are true games. They are not. They are a form of puzzle, like a crossword or a 5000-piece jigsaw or a mathematical brain-teaser. They are an entertainment, they are a pastime, but they are not a game. They have simply acquired the game sobriquet by default because they're like games and therefore can be roughly lumped together into the game classification. But we are too clever for that by half and will not make a similar mistake.

By the same token, we must exclude arcade-type games from our discussion here. Shooting aliens out of the sky, collecting hostages in your chopper, blasting enemy submarines—these are all a lot of fun and share some fairly important characteristics with true game-type games, but they primarily fall under the heading of games of technical skill, where the player's success or failure rests on his ability to manipulate items physically, in this case computer blips. The game of pool would fall into a similar category; one does employ game-like techniques against one's opponents, but ultimately one's success or failure rests on one's ability with a cue stick, a very technical skill. Pool certainly requires greater technical skill than "Choplifter," but I'm sure you'll agree that the analogy is apt.

Trying to define games is like trying to define man. No matter how fine a point you put on it, sooner or later someone will come along and knock you down. Man is not the only creature to walk on his hind legs, not the only creature who thinks, not the only creature who makes tools, not the only creature who can communicate symbolically, and so forth and so on.

The same difficulty of definition applies to games. I would like to say that a game is a structured cerebral conflict conducted according to a series of set rules by a certain number of players, eventually resulting in clearly defined winners and losers, but before I could get the words out of my mouth you could cite right off the top of your head a dozen perfectly valid games that do not fit that description. But it really isn't that bad a definition, and it certainly does cover the kinds of games we'll be discussing from this point on, so let's go with it.

For our purposes, the most important part of a game is, if you will, its soul. A good definition of the word game would include both tic-tac-toe and chess—both have the conflict, the rules, the winners and losers. But there is a world of difference in the complexity of these two games. The souls of these games are entirely different.

Let's look at tic-tac-toe for a second. This is a game of very simple rules and a very clear objective, connecting three marks in a row. Because the game is so simple, its soul is almost as obvious as its rules. Even most children light on the winning strategy for tic-tac-toe very quickly, which leads me to my axiom that in fact it is a game's strategy that is its soul.

If both players can connect with the soul—or strategy—of tic-tac-toe, this is what will happen. The player with the first move will take the middle square. The player with the second move will take one of the corners. From that point on, short of an incredible blunder, the game will always result in a draw. Now, if only one of the two players understands tic-tac-toe strategy—if only one player is in touch with the soul of the game—that player stands a good chance of winning quite often. Not every time—because there are so few possible moves it is easy for the ignorant player to stumble onto a good

one—but often enough. If neither player knows the strategy, it's anybody's game. And again, if both players know, short of a blunder it's always a draw.

You may not buy this, but all the above applies as well to chess, with only a difference of degree. Chess is an enormously complex game that no one has ever totally mastered. Some players have been phenomenally good, but none has ever been fully in touch with the soul of the game. So we take two players with equal (but not perfect) understanding of strategy, and what do we get? Well, we don't draw every time, that's for sure, because the possibilities for minute, subtle blunders are rife. But over the long term, playing both the black and white pieces, two equally matched players will roughly break even, each winning and losing about the same number of games (plus a lot of draws). With unevenly matched players, we find that as the difference in skill increases, the game more frequently goes to the better player until at some point the lesser player doesn't have even the slightest chance of a draw, much less a win. Chess allows for subtler degrees of skill than tic-tac-toe, but eventually it all boils down to the same thing: The player who has the better strategy wins; the player who does not, loses.

This is as good a time as any to differentiate between strategy and tactics. Both are important, but they are not the same thing, and as a prospective game expert you should be able to distinguish between the two. Strategy can be loosely defined as your overall plan for obtaining your major objectives. In chess, good strategy includes getting as much control as possible of the center of the board. Another part of good strategy is getting all your pieces out early in a strong offensive position while maintaining a solid defense as well. Other facets of your strategy might change as the game develops; white might go for a king-side attack while black goes for the queen side in some Sicilian variations, for instance. In any case, strategy is your grand scheme, and optimally your strategy is multifaceted and capable of adapting itself to new situations as a game progresses. Strategy is not to be confused with your goal, which is the capture of your opponent's king. That is your objective; your strategy is your plan for obtaining that objective.

Your tactics, on the other hand, are concerned with the immediate question of the conduct of the battle. You might be thinking, I want to control the center of the board—your strategy. But you have to decide how to do it, perhaps a vanguard of pawns flanked by both bishops. That is tactics. In World War II the Allied strategy in 1944 was to establish a beachhead in Europe through which to supply fighting troops to knock out Germany's thinly drawn Western defenses. The tactics employed to meet this strategy were the use of all those LSTs in Normandy, the parachuting of John Wayne behind enemy lines in *The Longest Day,* the blowing of this bridge, the blocking of that road, etc.

What does this all mean in terms of programming computer games? I want to communicate a feeling for the concept of strategy, because I will be throwing that word around quite a bit in the discussion to follow. Every game has a soul, a strategy. Some games are so simple that this soul is easily accessible, and therefore just about any player can achieve the perfect strategy. As the complexity of a game increases, it becomes harder and harder to find that perfect strategy, but it is still there somewhere. Many players can do little more than try to choose decent tactics in the heat of the moment—

this describes the vast majority of chess players. But within every game, somewhere, somehow, that soul, that perfect strategy exists. And as a game programmer it is your responsibility to get as close to that soul as possible.

One thing that should be obvious from all this is that it would be well nigh impossible to program a game with which you are not completely intimate. If you don't completely understand the strategy and tactics of a given game, there is no way you can get the computer to do what you can't do. And by understanding a game I mean that you know it inside and out. You know the perfect strategy, you know less-than-perfect strategies, you know the good tactics and the less-than-good tactics. I mean that you understand the game at every possible level. I do not necessarily mean that every time you play this game you always win, nor do I mean that you intend to program the computer always to win. But you have to be capable theoretically of both for one very simple reason: If you don't know your game well enough to do that, your program won't be very good.

## ALGORITHMS—INTEGRATING GAMES INTO PROGRAMS

The soul of a computer program is the algorithm, a term we promised to explain in detail when we reached this part of the book. So what, pray tell, is this mean-sounding animal? Putting computers aside for a moment, an algorithm is simply a formula that quantifies data in a meaningful way so we can act upon those data.

The human mind is capable of creating algorithms nonstop for every situation we encounter in life, and we never think about how this process is conducted. Somehow our minds are capable of quantifying just about everything we put into them, and one way or another giving us an answer, a choice, a course of action. For instance, I accidentally touch a red-hot frying pan. I now have innumerable options open to me. I could remove my hand from the pan, I could scream out in pain, I could hop back and forth from one foot to the other, I could sneeze, I could blow a raspberry. But my brain very quickly eliminates the supercilious responses, such as the raspberry and the sneeze. On a scale of 0 to 100, my brain instantly rates those two actions, and all others like them, a 0. Hopping back and forth, which I may indeed do shortly, is still rather nonessential, so the brain gives it about a 4 and holds onto it for further action later. But screaming out in pain and removing my hands are both involuntary reflexes, and both will get very high ratings. Screaming will get a solid 99, and you can be quite sure that my brain will have kept this high on the stack of responses and will get to it shortly. Moving my hand away will, of course, get a 100 rating, and that is the first thing I will do. And believe me, I will do it very quickly.

The point of this somewhat shaky analogy is to show how related data can be quantified by a processor, even though those data do not come attached with quantities clearly labeled. It is as if the brain assigns numbers to them according to some mysterious process that we'll call an algorithm, chooses the highest number according to that algorithm, and then acts. This is true

of other thought processes as well. Even as I sit here and type this book my mind is in its "write a book" algorithm. Every time I write one word my mind begins working on the next word, discarding those it does not like, keeping those it does like. Ninety-nine times out of 100 it will do this automatically, and my success or failure as a writer is entirely dependent on whether my mind's "write a book" algorithm is a good one or a bad one. In that one time out of 100 when the process is not automatic, when I get stumped trying to express myself, it is not that my algorithm stops working, it is just that a little more processing time is necessary before continuing. Please stand by while new material is sorted through and—yep! There's the right word, and on we go again. Now, anybody can sit down at a typewriter and knock out a series of words. (Who am I kidding? You know damned well I'm working at a word processor.) What separates a good writer from a bad writer is how each individual's "write" algorithm works. Some obviously work better than others. But we all have them, whether or not we choose to use them and regardless of how good they are. And this sort of analogy covers every thought process.

Our ultimate problem as game programmers is the creation of topflight algorithms that will quantify all the data pertinent to the game, giving us credible courses of action to follow in terms of the strategy and tactics of that game. This is the creative process of game programming. We must somehow translate everything we know about a game into a series of rigid computer statements that are completely arbitrary and not the least interested in gaming and end up with a program that not only will play a given game but also will play it well and that will be fun for the human participant in the bargain. To illustrate how this is done I have chosen the game of poker and written a program in which the human player squares off against four different opponents, each of which is played by the computer. Each opponent employs different tactics of play, and each opponent approaches the game with a different strategy. One of those players comes very close to playing perfect strategy, the other three are progressively less good at the game. But more about that later. The point is that within this very long and complicated program are the algorithms that control both the tactics of play and the strategies. The nice thing about poker is that it can be arbitrarily broken down into these two aspects very neatly for the sake of our discussions, because what one does with one's cards is one's tactics, and what one does with one's money is one's strategy. Not all games break down so neatly.

In the world outside of this book the term "algorithm" tends to be used much more freely to describe just about any solution to any computer problem. That would mean that in poker I have an algorithm to shuffle the cards, another to deal, and so forth and so on—which is fine, but I like to see the use of the word confined to the really major problems; and that's how I usually will intend it from now on.

There are no hard-and-fast rules for the creation of algorithms because algorithms tend to be unique solutions to unique problems. What you try to achieve in your game algorithms is the creation of a sophisticated player, an opponent to keep you on your toes. We will do our best in the rest of this book to give you a solid background in the techniques and thinking necessary to achieve this goal.

## GOING ALL THE WAY—WHAT MAKES A GOOD COMPUTER STRATEGY GAME

It is all well and good to say that you are going to program a poker game, but just how good a game are you planning on designing? I have seen so-called poker programs where the human player goes head on against the machine—in effect, two-man poker. Who ever plays two-man poker? In the real world poker is a contest among five to seven players; I chose five mostly because that number fits best on the 40-column text screen. There's no reason why you can't throw two more players into the program if you are so inclined and you've got the 80 columns to do it in. The point is, if the game isn't realistic right off the bat, it's already got three strikes against it. The primary condition necessary for the creation of a good strategy game is a total commitment on your part. If you really want to sit down and knock off a good poker or blackjack or bridge or cribbage or backgammon or whatever, you've got to steel yourself for a lot of work. It is not an easy job; working in your spare time, you could take from six months to a year to get a good one up and running. The satisfaction when you are finished will be enormous, and by the time you are finished you probably will have learned everything you'll ever need to know about programming. For me, these results have made this long, involving process very rewarding, and I admit quite frankly that as soon as I finish writing this book I plan to get reacquainted with my wife right after I find out what she named the baby.

By now you realize that it is impossible to program a game unless you know that game inside and out. This does not necessarily mean that you as a player are unbeatable at that particular game; even the best of humans tend to err. Since the computer does *not* err, if you've taught it everything you know it will most likely beat you over the long term, a fascinating prospect when you think about it. So pick a game you know and one you enjoy, because you're going to have to live with it for a long, long time.

As you develop your program, it is essential that "completeness" become your credo. As time goes on and you're working like crazy and you still don't seem to be getting anywhere, the temptation to begin cutting corners becomes very strong. One of the first programs I wrote when I was starting out was for seven-card stud. Every time the players got a new card the hands had to be reevaluated, which meant a cumulative effort geometrically progressing in complexity to figure out the value each time of what the players were holding. At the end I finally gave up, and after the last card was dealt I programmed a line that said:

I CAN'T SEE THE CARDS IN THIS LIGHT. PLEASE ENTER THE NUMBER VALUE OF YOUR HAND

followed by a choice from zilch to straight flush. How lazy can you get? Ultimately, this was quite a bum game. But after a couple of months I was tired of it and ready to move on. My time programming wasn't wasted—I learned a lot—but when I was finished I didn't have anything to show for it. Be prepared for similar frustrations.

The term "whistles and bells" is used a lot by professional programmers, and it's a concept to keep in mind in your designing. Whistles and bells in a program don't do anything, they simply blow and ring to put a little pizzazz

into things. In "Space Derelict!" the slowed-down speed of some of the messages transmitted in inverse type falls into the whistles-and-bells category. Sound effects and nonessential graphics can be whistles and bells, and they're usually nice to have. The poker game in this book is not graphic because, as I said earlier, graphics are just too major a subject and we don't have the space to give them their due. If you really want a graphic deck of cards, however, I will give you an elementary one a little later. Anyhow, keep in mind the idea of whistles and bells, and incorporate them into your programs freely. Don't get too carried away with these nonessentials, but a little bit here and there can make for more fun in the playing.

You probably will find after you have finished your program that there are some slow spots in it, where the programming was a little bit loose so you find yourself sitting around waiting for things a little longer than you'd like. Once again, you may be so tired of the program at this point that you simply will accept its pokeyness, but if at all possible you should keep at it and take the necessary steps to get it streamlined. We will be discussing some of these steps as we go on, although I was forced to skimp on a few of them in poker to keep the program readable. But I'll show you the ways speed can be improved, or if not improved at least hidden so no one will realize how slowly the processing is going. It's all right to be sneaky; if you get away with it, great, and if you don't, people will nonetheless admire your cleverness.

Finally, despite everything I've said to the contrary so far, it is important to remember that unless you are a full-time programmer, it is practically impossible to create a program as good as you'd like. So part of your training will be to learn how to create workable algorithms that, even though they do cut an occasional corner, play well regardless. This requires a bit more sneaky creativity, but I've said it before and I'll say it again: If it works, do it.

## THE GAME OF POKER

Why choose poker as an example of strategy games for inclusion in this book? A good question, and I'm glad I asked it.

First of all, knowledge of the game of poker is pretty widespread, and I knew that even if you weren't a dyed-in-the-wool poker buff you would know enough about the game to be able to follow the discussion of the program. But the odds are that you *are* something of a poker buff: I have read some vague statistics that indicate that eight out of seven people in America play poker a minimum of six Fridays a month—or something like that. Poker is quite definitely the national card game, so what better subject could we possibly attack?

There is another advantage of poker, and that is that there are enough variations to choke a horse. So after you've absorbed the program in this book you can adapt it to your own favorite poker games; not as malleable as the adventure module, perhaps, but still adjustable.

Finally, I plain wanted to write this program. I enjoy poker and lately have lost the proximity of my regular monthly game. So I've re-created that game exactly, and now it's Friday night whenever I want it, with the chips stacked neatly, the beer chilling in the refrigerator, and all the players eager to get

started. That's a big part of what computer gaming is all about—the willing player, whenever you want it.

Besides all of this, there is something within me that says that if you can master all the Basic techniques we'll be using in this poker program, you'll be able to master just about everything that will ever come up in any kind of strategy game. The ultimate idea is to prepare ourselves for any programming challenge we might face; whereas the adventure module is just there, waiting for us to fill in the blanks, for a strategy game (other than a variation on the poker theme) we start with a tabula rasa: you'll have programmed poker, so now try Parcheesi. The important thing here will be to follow my thinking rather than my program.

The game of poker, while simple in some ways, is quite complicated in others. Perhaps that is its attraction as a popular pastime. It's easy to learn and play, but it contains incredible nuances, and it's a lot of fun. Wouldst that all games fell into that category! In terms of game theory, poker adds a new twist to what we were discussing earlier in this chapter—the element of chance.*

In a game like chess, there is no question that the person who plays the better game will emerge the victor. The only factors involved in the play are the abilities of the two players, and the game rests on their strengths and weaknesses. But in poker, the deal of the cards is a big factor. Although it is not inconceivable that you could beat a good hand with a bad hand—that's what bluffing is all about—in normal play over the long term good hands tend to beat bad hands more times than not. The "straighter" a poker game is, the truer this is. In five-card draw, it is about as true as it can get. In Hold 'em, which is played in the World Series of Poker in Las Vegas, this is much less the case. Hold 'em, a version of stud in which the players share three up cards, is very much a game of betting strategy and bluffing. Since so much information is available to the players and since they even share the majority of the cards, all the play concentrates on the hole cards, the ones the other players can't see. There is no draw; one simply plays the cards one has. Very much a betting game.

Straight five-card draw, jacks or better, requires the ability to understand what the game of poker is all about and also the ability to bet a better game than your opponent. Both skills are roughly equal in importance, and one of the joys of poker is that the latter skill changes with the opponents you face, and even changes over time if you always face the same opponents. No two players play the same way; much of one's success at the game is measured by how well one can read the other players. Even if you play with the same opponents year in and year out, each player will continue to grow and change in ability. Again, some of the infinite attractions of the game.

The first half of your ability to play decent poker requires only that you understand and trust implicitly the idea that every time you flip a coin the odds are 50 percent that it will come up heads and 50 percent that it will come up tails. This is what we all learned in high school as probability. If a coin comes up heads 100 times in a row, the odds are still 50 percent that it will come up heads the 101st time (assuming the coin isn't rigged). This is

*I can't resist my favorite W. C. Fields line, as he sits down to play poker with a friendly Indian. Indian: "Is this a game of chance?" W.C.: "Not the way I play it."

true; we all know this—and we'd all play better poker if we remembered it all the time. "Luck" has nothing to do with it. Just because we've been doing well all night does *not* mean that our odds of drawing to an inside straight are improved. The odds of filling an inside straight are *always* 4 in 47—four chances of finding a card of the right value in the 52-less-5 that comprise the universe outside the cards we hold. Roughly 1 in 10, pretty shaky odds at best. The good poker player avoids the inside straight like the plague. Let's take each possible poker hand and look at the possibilities.

First there is the zilch hand, no pair, no more than two of any one suit. The best we can do is keep the two highest cards and hope for the best. The odds here are not so bad, actually. Given that there are three each of the two cards we're keeping still outstanding in the deck, we have a 6 in 47 chance of hitting a match. (For this discussion I'm going to forgo the complications that might ensue—in this case, for instance, you *could* draw two of one and one of the other for a full house, or three of one for four of a kind, and so forth, but the possibilities for this are fairly insignificant and are for the most part constant for all the hands. Let's try to keep it simple.) About 1 in 8 is a tad high but not ridiculous. The real problem with this hand is that you are holding nothing and shooting entirely at birds in the bush. How much money should you bet on such chimerical possibilities? None, if you're going into my wallet.

Next you have your three flush. You have a 10 out of 47 chance of getting one match, *compounded* with another 9 out of 46 or the second match. Somewhere around 25 to 1, since you have to multiply the two odds figures (that's another thing they taught in probability class, that when two possibilities are dependent on each other, the probability of getting both possibilities is not the sum of the two possibilities but the product of their multiplication). You've got to be crazy to play a three flush, and a three straight is about the same, so you have to be just as crazy for that one.

A four flush, of course, gives you 9 out of 47 possibilities straight off, which isn't bad, especially compared with the payoff of a very high hand if your ship comes in. A four straight, not inside, is a bit higher, 8 in 47 (filling either end), but still worth the push for the rewards of success.

Hands of a pair or more tend to play themselves, with two exceptions. First, there's the low pair when you also have a four flush or four straight (for instance, the 7 of spades, the 7 of clubs, the 2 of clubs, the queen of clubs, and the 5 of clubs). What do you do in this situation? Well, your chance of improving the pair is higher than the chance of improving the four flush—you could get either another pair or three of a kind with more likelihood than that fourth flush card. How do you play it? How much of a gambler are you? It would really depend on how the betting has been going and what you feel the other players are holding and what you'll need to beat them. Very subtle, very difficult. The second exception to pairs playing themselves is the "kicker"—i.e., holding an ace when you've got a pair or three of a kind. Suffice it to say that your odds are slightly lessened when you hold a kicker, but you may make up for this in the value gained of confusing your opponents as to what you're holding. Again, a fielder's choice, you vs. your opponents.

Despite the logic of what to do and not do vis-à-vis the odds, not all players follow the party line all the time, either because they don't know, or

because they are trying to trick their opponents. The first thing we have to consider when writing a poker program is how well the game will play for the person sitting at the keyboard. One of the things we can do in five-person poker is put four opponents up against the player and give each of these four opponents different abilities, which is exactly what I've done. This goes back to the basic concept of making a computer game fun for the player. If all four computer opponents played exactly the same way, it wouldn't take long for our player to figure out what the computer will do in any given situation, since he has four examples of it in front of him all the time. But if each opponent plays differently, not only is our player figuring out how best to play his own hand, but also he now has to unravel four distinct opponent "personalities." The value of the game is multiplied accordingly.

But all of this is concerned with what we've called the tactics of poker, the playing of the cards. What about the strategy, or betting? Well, here we have to take a somewhat different approach, as you'll see when we get down to the brass tacks of the programming. Again, all four players play differently, their betting strategies roughly analogous to their tactical abilities. But here we have to be much more conscious of the human player. Ultimately our betting strategies are static: They are etched into computer stone and, unlike the tactics that are etched into the same stone (a player can never be sure of the unseen cards in the opponent's hand), once a player knows all four computer strategies, he might be able to construct his own strategy to beat the program regardless of tactics. If an opponent always bluffs at a certain time, if an opponent always bets a certain unique amount with a given hand, if an opponent always folds when the player bets a certain amount, the player gets an upper hand that can destroy the fun of the game. So we prepare for this as best we can by making our four strategies as complicated as possible. However, we must face the inevitable, that any program we create is unlikely to stand up against the determined player forever. But we do our best, and that's what it's all about.

# 8

# PLANNING THE PROGRAM

## STRUCTURED PROGRAMMING?

There are two words that can strike terror into the hearts of the most valorous computer programmers. These two words can be seen to represent everything from the Holy Grail to the shackles of oppression that must be broken. For true believers, the structured program in Basic is the unattainable; even the best of hackers will coyly comment that the program they've just written is pretty good but not really structured (even the IIe manuals are not exempt from such *mea culpas*, and these are the people who are selling us our machines in the first place). Academics will insist that structured programming is more important for its platonic value than as a practical aid. Iconoclasts (sometimes but not always including this author) say that if it works, do it—but will think in terms of structured programming when the situation calls for it. And in the business world, in many situations structured programming is a given because the nature of the beasts preclude any alternative. But what do these two words mean, to arouse such varying yet universally strong reactions?

As you know, the digital computer operates entirely in terms of on-off. Electrical impulses flash beneath your keyboard at infinitesimally brief speeds, and the processor performs certain operations on the basis of the content of these impulses. These impulses are either on, an arbitrary value of 1, or off, an arbitrary value of 0. The binary number system, which contains nothing but 0's and 1's, is the mathematical measurement of what the computer is doing. The computer's native language is binary.

Binary numbers look like this: 10110010. For the sake of simplicity (trust me), someone once decided that numbers that look like this—2C, FF, A0—are easier to handle than binaries, so the hexadecimal number system is the

one that the computer most regularly speaks to human beings. Fortunately, for us real dummies, the computer can speak decimal numbers as well, but that's two generations away from the native system, which means that occasionally translation problems can arise and lead to difficulties. For instance, 64K does not really translate into 64000 bytes, but we all know what we mean and we're close enough, so nobody really faults us for our third-generation translated approximation in this case. In other, more technical situations, this translation could be a problem (but fortunately one that need not concern us here).

Computers do *not* speak Basic, COBOL, Fortran, Logo, or any other so-called computer language. These languages tend to be cooked up in the basements of name-brand universities or the like, and their purpose is to provide a common ground for communication between the computer and the human. The more humanlike that language is, the "higher" the level of the language. To get really down and dirty, the lowest-level language is binary machine language, which no one really speaks except the computer. But with the use of something called an assembler a human can come reasonably close to binary functions insofar as they are translated into a language of hex numbers. 6502 assemblers use key words such as JMP and LDA (which humans can learn to understand) and then create (or assemble) hexadecimal machine-language programs from these key words. Basic, on the other hand, is a high-level language, and in many cases it reads like plain, simple English. You don't have to know anything about programming to understand the line:

PRINT "CALL ME ISHMAEL."

or even:

X = 100: PRINT X/20

IF, AND, OR, NOT, GOTO, END, CLEAR, etc., are all good human words, at least for English speakers, and in many cases program statements that incorporate these Basic keywords are easily understandable to the uninitiated.

But how does Basic work inside your computer? Well, there's an imaginary little magician inside there known as an Interpreter. When you write an Applesoft program the Interpreter translates each line into a series of hex numbers; for the most part, this is a literal translation, and the result is most definitely *not* a machine-language program: It is a Basic program written in hex numbers, period. When the program is run, the Interpreter takes each line as it comes, turns it into a machine-language operation the computer can understand, the computer performs the operation, and then the Interpreter reads the next line, and so forth. From this you ought to be able to see why a Basic program runs more slowly than a machine-language program; each line first has to be translated, the process continuing one line at a time, whereas the machine-language program simply performs the operations without the Interpreter in the middle.

There is one way of getting around this problem of translation slowness, and that is to use something called a compiler. Compilers are common in large businesses; a business-oriented language such as COBOL, for instance, is actually a compiled language, as compared to Interpreted Basic (although compilers do exist for Basic programs, they are quite expensive and perhaps

not all that useful for hobbyists). A compiler is a program that literally rewrites a high-level-language program into a machine-language program (it does some other things as well, but they are beside the point to us—we're simplifying what follows a bit to zero in on our own concerns). After the computer people in your company spend three years futzing around writing a payroll program in high-level COBOL, they compile it. Then the compiled program, now in speedy machine language, is the program actually run on the mainframe computer. Since this program is the one responsible for making sure your paycheck arrives punctually on your desk at its appointed time, you can see why speed can be useful. It might take 24 hours to run this program and get all those checks out every two weeks, and that's the fast version. If the program were not compiled, you would probably be paid your weekly check every three months or so, which you might find inconvenient, to say the least.

Can you imagine that payroll program, that is, the uncompiled COBOL version? You've seen those business printouts of hundreds of pages of code; you know how data processing works in business where one person works on one part of a program, another person on another part, and so on. Can you imagine the complications not only of writing but also of debugging this monster? Hold that thought for a minute.

Structured programming has many purposes. The concept arises in part from the needs of professional business programming. What is the best way to write a program that will lend itself to all the problems writing these monster programs can lead to? Since there are usually about 100 different ways in any language of writing a program to perform the same chore, and we have about 100 different programmers working on the same program, how do we set a plan for the work so that everything each individual programmer is working on fits in with the other programmers' work? We need an overall structure from which the individuals can branch and do their own work however they're doing it, a structured program wherein one big monster is broken down into manageable parts so we never really have to face the whole monster at once.

The same holds true for us when we single-handedly plan a major program. Even though I am one person with one style, this program is going to take me months to complete. Once I'm halfway through it I'll have long forgotten what I was doing at the beginning. If I attempt to create one entire monster all of a piece, that monster will ultimately devour me. But if I structure my program into manageable pieces, I will never have to face the monster all at once, and I will beat that demon and reign victorious for all to see! Or something to that effect.

Further, there is the issue of sheer efficiency for its own sake. Structured programming breaks down the computing process into meaningful segments based on what exactly is going on in those segments. Some high-level languages, such as Pascal, are actually structured programming languages, which Basic is not. So when uninitiates look at a Pascal program, not only can they recognize and understand some of the English commands as they can with Basic, but they can also see to some degree just how the program works: its total structure, its processing flow, is fairly easy to comprehend even without a knowledge of the language. This is definitely not true of

Basic, where although the words might make sense, there is no guarantee we'll understand how a program works just by looking at it (even if we've written it ourselves).

But let us backtrack for a minute. With a noncompiled language such as Basic we have the additional problem of that slow process of line-by-line interpretation. To be good programmers not only do we need to understand the structure of a program but also the structure of our language. Since we can't compile our work, what can we do to speed up the processing? Programming structure and language are two different things, but we must think of them simultaneously if we wish to create satisfying Basic programs, especially games. It is no fun to sit in front of the computer twiddling our thumbs while we wait for a particularly convoluted piece of programming to process. So we'll no longer differentiate between the two structures. We'll just lump everything into the concept of structured programming and give the academics another reason to throw this book out the window. In unstructured Basic, structured programming is what we make it.

## SOME BASIC PRINCIPLES

For us, the simplest definition of a structured program is one consisting of a series of small subroutines, each subroutine performing one complete part of our overall task. The skeleton of this kind of program might look like this:

```
10 GOSUB 1000: REM INTRO
20 GOSUB 2000: REM INITIALIZATION
30 GOSUB 5000: REM COLLECT USER INPUT
40 GOSUB 10000: REM PERFORM CALCULATIONS
50 GOSUB 20000: REM DISPLAY RESULTS
60 GOTO 10: REM DO IT AGAIN
1000 & ff INTRO and RETURN
2000 & ff INITIALIZATION and RETURN
5000 & ff INPUT and RETURN
10000 & ff CALCULATIONS and RETURN
20000 & ff RESULTS and RETURN
```

Somewhere in the program, in this case lines 10–60, is a series of commands directing processing to the various subroutines, sort of like a traffic cop directing the flow of the program. And while the same data might be used in more than one of the subroutines, each subroutine performs an entire and unique operation. The value of this is twofold. First, when you sit down to program, you have to handle only one subroutine at a time. Second, and perhaps more important, when you debug you can extract each subroutine and concentrate on it rather than having to find your way through the whole program. After you've got one section completely oiled you can move on to the next. Even when you have the whole thing up and running and those last few secretive little bugs finally pop to the fore, this compartmentalization will make it easy for you to recognize exactly where in the program the problem lies so you'll know where to go about looking for it. Another name for this sort of structure is "top-down" design.

The first step in planning a program of this nature is to make a list of all the things you will have to do in the program. Once upon a time even Basic programmers used to draw flow charts with pretty little boxes and circles and diamonds, but I understand that this practice is falling by the wayside, and I'm sure that I've never seen any hobbyist drag out a template and start drawing. A nice legal-size yellow pad is my own favored scratch sheet, and whatever you find convenient is what you should work with. Let's try an outline for tic-tac-toe first. If you're really feeling energetic, don't read what I've written here until you've written out a little outline of your own.

```
GOSUB INTRO—EXPLAIN RULES
GOSUB INITIALIZATION OF VARIABLES
GOSUB DRAW T-T-T GRID ON SCREEN
GOSUB DETERMINE WHO GOES FIRST
GOSUB COMPUTER'S MOVE or
GOSUB PLAYER'S MOVE
GOSUB DRAW MOVE ON GRID
GOSUB FIGURE IF GAME IS OVER
IF GAME OVER THEN GOTO ENDING
IF GAME NOT OVER THEN GOTO NEXT MOVE
GOSUB ENDING
CLEAR GRID AND GO BACK TO DETERMINE WHO GOES FIRST
```

The flow here looks simple enough. First an introduction subroutine followed by initialization. Then the grid is drawn on the screen. Then we determine who goes first; this could either be at the player's option or else simply alternating back and forth between player and computer. With this information in hand, we know whether to take the GOSUB where the computer makes a move or the one where the player makes a move. Whichever it is, next we draw the move on the screen (our player's move subroutine would have covered the possibility of the player making an illegal move). Then we figure if the game is over. If not, we go back to the appropriate subroutine for whoever has the next move. If so, we go to the ending, perform the necessary congratulations or raspberries, and start again.

In the actual program, all this movement would be traffic-controlled just like our skeleton program, with all the real action taking place in the subroutines. We would then proceed to write one subroutine at a time. You could either debug each subroutine as it is written, or you could wait until the end and debug each routine with the whole already laid out in front of you; in either case you would still be concentrating on the individual subroutines. It all looks very easy, doesn't it? In a way it is, and that's the whole point of structured programming.

One thing we're overlooking, however, is a problem I mentioned in the first section of this chapter. You'll remember we talked about the Basic Interpreter and how this quite wonderful little marvel in fact kept slowing us down, taking every line of programming one at a time. And, too, you'll remember from the first part of the book that every time a GOTO or GOSUB is encountered in a program, the processor goes all the way back to the beginning of the program until it finds the correct line to branch to. This, of course, is the fault of the Interpreter. It's one thing to talk about what's really

going on in a Basic program, and another thing altogether to look at one. So let's show and tell for a minute.

```
*800.84F

0800- 00 1C 08 0A 00 86 41 24
0808- 28 32 33 29 2C 43 4F 24
0810- 28 31 36 29 2C 43 4F 28
0818- 31 36 29 00 27 08 14 00
0820- 86 44 28 31 37 29 00 2D
0828- 08 1E 00 89 00 33 08 28
0830- 00 97 00 4D 08 32 00 B0
0838- 31 35 30 30 30 3A B2 20
0840- 49 4E 54 52 4F 44 55 43
0848- 54 49 4F 4E 00 53 08 3C
*
```

What do you think of that? These are roughly the first five Basic lines of "Space Derelict!" the way the computer sees them. Can you believe we wrote that ourselves? Well, in fact, we didn't; the Interpreter did the work for us. Let's look at it.

This is a listing of the machine addresses from $800 to $84E (hexadecimal numbers are prefaced with dollar signs—this is because they are more expensive than decimal numbers). Unless you go out of your way to do otherwise, all Basic programs will begin at address $801, and they all begin after a byte 00. This is something we accept unquestioningly, because that's the way it is. The next two bytes in this listing are 1C 08. These two numbers are, in fact, the number of the address in memory where the *next* program line listing will begin. This address is $81C (or $08 1C—the Apple reads the bytes backward, which is why the Apple is the computer and you're the human). Immediately following this information we see the two bytes 0A 00. Again this is backward, and this is the first line number listing you would see in the Basic program; in this case the number 10, the decimal equivalent of $000A. Let's skip then to $81C—this address is represented by the fifth pair of numbers on the line marked at the left as 0818 (hexes run like this: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 1A, 1B, etc.). Again, we get the number of the next address, $827, followed by the program line number of this listing, $0014 (20 in decimal). Let's skip again to $827, which gives us the next address as $82D. $82D tells us to go to $833. $833 tells us to go to $84D, $84D tells us to go to $853, and so forth. Whenever you do a GOTO or GOSUB, you've got to go through all that; that's how the computer knows where the lines are. Certain information about your Basic program can be and is stored in other places around the computer; addresses $67 and $68, for instance, contain the bytes directing the computer to the beginning of an Applesoft program. But as you can imagine, there is no shortcut for GOTO and GOSUB searches: There is no way that the program line equivalent of every line could be accessed outside the actual program. It would be redundant and would take up half again as much memory as the program itself. So we have to get

around this limitation, and the best way to do it is never to use GOTO and never to use GOSUB. Impossible, you say? All right, we'll compromise. You can use GOSUB, but we'll put the more frequently used GOSUBs at the beginning of a program so we won't have to go through all this rigamarole at every access. Of course, you still can't use GOTO, but more about that in a second.

Let's go back and look at our tic-tac-toe. Some of these subroutines would come up only once in a program, but others, such as the figuring of moves and figuring the state of the game, would keep coming up again and again. So what we would do is put these frequently accessed subroutines at the beginning of the program, where the Interpreter can get to them as quickly as possible. Makes sense, doesn't it? We'll talk more about this when we see how it was done in the poker program. The logic of it should be clear, however, at this point.

But what about those GOTOs? Hell, "Space Derelict!" was positively a breeding ground for them, and it ran fine. Very true, but the number of calculations in that program from the moment a player enters his input until the results are seen on the screen is very small. Computers are by nature fast, remember, so sometimes we really can't see the delays we're creating. But in a game like poker, or most likely any strategy game you create, you will be setting your computer on some fairly long and involved calculations, and there will be a noticeable pause every now and then while the computer does what you are asking it to do. The point of minimizing GOTOs is to keep these pauses down as much as possible. Also, the use of GOTO can lead to what is so ingloriously referred to as spaghetti logic, wherein a program resembles a plate of the same, and it becomes very hard for anyone, including the programmer, to figure out what he is doing or where he is going. "Space Derelict!" does have its share of spaghetti logic, but in that simple sort of program, where there isn't that much spaghetti in the first place, it doesn't matter very much. But in a complicated program, you want your logic as clear as possible. The normal temptation is to program yourself into a corner and then program yourself out of that corner, whereas what you should do when you get into that corner is simply throw away what you've just done, rethink it, and start again. So we can't, of course, dispense with GOTO altogether, but we can use it carefully.

One thing always to be on the lookout for is the mob and the leper at the turn in the road. There is a sign at this turn: "Lepers turn right!" We want to make sure we don't have the mob going right, as this will only slow them down while they read the sign and try to figure out which direction to go in. Translated into programming, in the flow of what you are doing you will always be breaking down your information into patterns and variables, and after the breakdown is finished some of these variables will go in one direction while others will go in another direction. You want to redirect the lepers, the minority, while the mob continues along on the most direct route. For example:

```
10 INPUT "PICK A NUMBER FROM 1 TO 10";X
20 IF X < 10 THEN 100
30 PRINT "YOU PICKED THE NUMBER 10"
40 etc.
```

This is an example of meatball thinking, which is this book's answer to

spaghetti logic. What we're doing here is directing the mob while the leper goes through in a straight line, whereas the odds of getting the leper are only 1 in 10. Doesn't the following make much more sense?

```
10 INPUT "PICK A NUMBER FROM 1 TO 10";X
20 IF X = 10 THEN 100
30 etc.
```

In this case our sign at the turn is directed at the leper, while the mob moves along in a straight line. This analogy may be a tad strained, but I like it, and however you choose to visualize it, it is important to remember.

There's one other thing to keep in mind, and I want to go back to the machine addresses given above. In the Basic "Space Derelict!" program we have this:

```
30 TEXT
40 HOME
```

This is what that looks like in hexes, starting at $827:

```
2D 08 1E 00 89 00 33 08 28 00 97 00
```

Eight of these 13 bytes are now devoted to the business of telling the Interpreter where we are and where we're going. But what if we were to rewrite the Basic lines as:

```
30 TEXT : HOME
```

Then we'd get the following, starting again at $827:

```
2F 08 1E 00 89 3A 97 00
```

In this case we have reduced our content to eight bytes, only four of which contain addressing information. We have killed two birds with one stone: We have taken up less space in the program, and we have put in fewer addresses for the Interpreter to sort through when the necessary GOSUBs and GOTOs are encountered. The message is obvious: If you can put multiple statements on a line, do so. The value is enormous, and we'll be doing it as much as possible in our poker game.

## THE POKER FLOW

Now let's get down to the business of the poker program. The first step was a work-up of just how the program would flow; that work-up was created in about three minutes on a piece of yellow legal-size pad, and looking back on it now that the program is finished, it is surprising how well it held up. But that was the point of it in the first place, so maybe it's not so surprising after all.

When working out a program flow, simply picture in your mind all the progressive steps of the game you wish to program. Sometimes you'll end up with a very vague outline that won't take into consideration some of the unforeseeable thorny problems; at other times you'll find yourself anticipating just about every complication. Either way, you'll face the problems in actual programming sooner or later, so it doesn't matter much how detailed your preliminary outline is at this point. Do it however you feel the most comfortable.

Five-card draw is a relatively simple game in terms of moves. The players are dealt their cards, they bet, they draw some new cards, they bet again, the winner takes the pot. Seven-card stud would have been a much more com-

plicated program because the players get dealt three cards, bet, get another card, bet, and so on up to seven. After each card is dealt, the program has to reanalyze the hands, and the programmer has to know all the moves and variations from three cards up to seven—a lot of work and a lot of programming. Also, in our five-card draw we are going to insist on jacks or better to open; this means that we need not concern ourselves with creating betting algorithms in the face of sheer guesswork. When you play draw as "anything opens" you end up with a fairly wide-open game. To program this you have to create an algorithm that not only would evaluate a hand but also would decide whether to open on that hand. In real life the factors guiding that choice are complex. If you have a good hand but are sitting too close to the dealer's left, you might pass in hopes of sandbagging—raising someone else's opener. This sort of decision rests almost entirely on player psychology, which is why it becomes very difficult to computerize it. But we're avoiding it entirely with our jacks-or-better game.

This is how my flow looked originally:

INTRO
SEED RANDOM NUMBERS
INITIALIZATION OF VARIABLES, HOUSEKEEPING
FIGURE WHO DEALS & WHO'S STILL IN GAME
ANTE
SHUFFLE
DEAL
FIGURE HANDS
ASSIGN VALUES TO HANDS
FIGURE IF THERE'S AN OPENER
IF OPENER THEN SKIP TO FIRST ROUND OF BETTING
WHEN NO OPENER HAVE SPECIAL NO-OPENER ROUTINE AND GO BACK TO FIGURE WHO DEALS
FIGURE BETTING PATTERNS
FIRST ROUND OF BETTING
GET DRAW CARDS
FIGURE HANDS
ASSIGN VALUES TO HANDS
FIGURE NEW BETTING PATTERNS
SECOND ROUND OF BETTING
DECLARATION OF HANDS
FIGURE WHO WON
CLEAR VARIABLES AND DO IT AGAIN

For the most part, this is self-explanatory. The first step should be familiar by now, as should be the third; we'll talk about the second one in the next chapter. You can count on these three in any strategy game with a random factor (in this case our deck of cards), and two out of three any other time. Some of the steps that follow, while still somewhere in the final program, turned out to be unnecessary; the fourth step, figuring out who's dealing, is one of these, and when I finally attacked it this became a very simple matter and needed only a line or two and not a whole routine. The round of betting where there is no opener ultimately became absorbed into the standard first round of betting when I started streamlining. The step "figure hands" means coming up with a variable for each possible hand that would stand us in good

stead when we came to the draw. For example, if a hand was "P23" that would mean we had a pair, the second and third cards of the five in the player's hand. Assigning values to a hand meant simply giving the hand a number in the hierarchy of hands from zilch to straight flush; our "P23" eventually would have a value of 5 (we'll go into all this in detail later, so don't worry about it now).

This list on my yellow pad was all I had to go with as I began programming. The first thing I did was translate this list into a series of GOSUBs, as in

```
10 GOSUB INTRO
20 GOSUB SEED
```

etc. Then I began writing each subroutine, filling in the blanks, trimming here, cutting there, streamlining, rethinking. This is how the traffic controller in "Five-Card Draw" finally ended up (and here's where to begin typing if you want to save the program—the *FIVE-CARD DRAW* legend is what to look for):

***Five-Card Draw***

```
10    GOTO 10000
10000 GOSUB 60000
    : REM   INTRO AND SEED
10010 GOSUB 50000
    : REM  HOUSEKEEPING
10020 GOSUB 6000
    : REM   SHUFFLE AND ANTE
10030 GOSUB 18000
    : REM   DEAL
10040 GOSUB 90
    : REM  FIGURE HANDS
10050 GOSUB 8000
    : REM  FIRST ROUND OF BETTING
10060 GOSUB 19000
    : REM   CLEAR SCREEN
10070 GOSUB 2000
    : REM  GET DRAW CARDS AND FIGURE HANDS
10080 GOSUB 5000
    : REM  SET NEW PATTERNS OF BETTING
10090 GOSUB 19000
10100 GOSUB 9000
    : REM  SECOND ROUND OF BETTING
10110 GOSUB 19000
10120 GOSUB 27000
    : REM  DECLARATION
10130 GOSUB 28000
    : REM  FIGURE WINNER
10140 GOSUB 29000
    : REM  CLEAR THE DECKS AND DO IT AGAIN
```

Right off the bat you can see that this traffic-cop routine does not reside at the beginning of the program. The first line of the program is line 10, sending us way up to 10000. What we've done here is put our most active

programming down at the early lines; 10000 is roughly in the middle of the program. This is a normal structured programming procedure in Basic, the actual placements depending on exactly what sort of calculations you are asking the computer to perform. Needless to say, the values of the numbers of the program lines are meaningless to the Interpreter; it takes no longer to read the bytes containing big numbers than it does little numbers. That is, if your program goes 10, 20, 80, 9090, 10000, 10010, and so forth, it takes no longer to go from 80 to 9090 than it does from 10 to 20; each decimal number is still represented by two bytes of hex, and that is all that matters. The usual practice for convenience's sake is to begin all new subroutines at thousand or hundred intervals, depending on the size of the program.

As you can see, the introduction and seeding are now combined at 60000, followed by housekeeping at 50000. Since these two routines are encountered only once, at the beginning of the games, they are way up in the listing, saving the prime real estate for more frequently accessed subroutines. The two once-separate acts of shuffling and anteing are also now combined, a little better situated at 6000. The dealing of the cards, which is actually the formatting of the screen into five separate hands, happens at 18000; speed is no factor here, as we want to deal the cards fairly slowly (or at least slowly in terms of computer time).

The first big gun, way down low at line 90, is the routine for figuring hands, which is quite slow because it forces us to use numerous GOTOs, and to put it simply, the bad hands take longer to process than the good ones. This routine goes step by step, eliminating all the possibilities; obviously, the worse a hand is, the more possibilities that must be eliminated. So this one gets the prime real estate in the program.

The first round of betting (whether or not there's an opener) is next, at 8000. The clear-screen routine actually does not clear the whole screen, just parts of it. We'll be doing some tricky screen formatting in this program, which, if you can master it, will make you a master formatter, a skill not to be valued lightly.

Getting the draw cards and refiguring the hands become one subroutine, again quite low at 2000. This one is a bit swifter than the first hand figuring, because now at least some possibilities have been eliminated. But again, still prime real estate. Hard on the heels of this GOSUB is the routine for setting the new patterns of betting (the old patterns will have been set in the figure-hands routine at 90—again, your streamlining at work). 19000 clears (part of) the screen again, followed by a second round of betting, another clear, the declaration, the winner, and the clearing of the decks to do it again.

We'll talk more about the rationales for locating the routines when we discuss the routines themselves. For now, the last thing to point out is that apparently (although only theoretically) this all could have been done in one line, thus:

```
10000 GOSUB 60000: GOSUB 50000: GOSUB 6000: GOSUB 18000: GOSUB 90: GOSUB 8000: GOSUB 19000: GOSUB 2000: GOSUB 5000: GOSUB 19000: GOSUB 9000: GOSUB 19000: GOSUB 27000: GOSUB 28000: GOSUB 29000: GOTO 10000
```

Ignoring the fact that there are some exceptions to this flow (again, to be discussed later), why don't we do it something like this? We could if we

wanted to. If we were really that hard up for space in the program, this sort of thing would not have been impossible. But it would have been extremely confusing. A traffic cop in real life stands in the middle of the street where you can see him clearly so you can follow his directions exactly, and that is what we're trying to do here. A tight program does not mean incomprehensibility (or at least it shouldn't, although sometimes it can, which is a problem we must strive to avoid). Clarity is just as important as flow speed, so don't get carried away with this stuff. Make it easy to understand, even if you're the only one who's ever going to have to understand it. Confusing programs are what give programming a bad name.

# 9

# SETTING IT UP

## RANDOM, SCHMANDOM—AND HOW TO UNSCHMANDOMIZE IT

If you believe everything you read in the manuals, the Apple has a built-in capability for creating random numbers, but if this is the case, you could have fooled me. Let's look at the evidence.

I've already mentioned my late, unlamented seven-card-stud program. I worked on this program, for better or worse, for many months, and when I finally had it up enough at least to throw *something* onto the screen, I began debugging it. Eventually it boiled down to this: I would sit down, run the program, and start playing until the bugs started biting. As time passed, I was getting through more and more of the deal, but I began to notice that something was fishy. I started keeping some records of the play and noticed that every time I turned on the computer and proceeded directly to running this program, I always found myself getting dealt the exact same hand. I then started experimenting and, sure enough, I found that no matter what I did, whenever I turned on the computer and tried to generate random numbers, I always got the exact same series—the exact same numbers in the exact same order. So random wasn't random after all. And no one ever said a word about it in the manuals.

Needless to say, you can't have games that incorporate random factors if those factors are always the same every time you play. Whenever the player sits down he'll find himself facing the exact same patterns, and this is true no matter how those random factors are incorporated. This is much more obvious where the player receives a hand of cards, say, than when he throws the dice in a game of backgammon, but even in the latter case it will become obvious sooner or later. But there is a way around this, a technique so nice

that its uses are limitless. It will be the niftiest thing you'll pick up from this section.

The first subroutine in our poker game is the first subroutine in any game, the introduction. The length and complexity of your introduction are entirely dependent on the nature of your game. If, for instance, the game is one of your own invention (something we'll discuss toward the end of this book), you will obviously need very detailed instructions. However, if you're programming a game where you can expect the player to be familiar with the rules, you needn't get carried away. If you tackle bridge, for instance, you could fill up your entire RAM trying to explain how the game is played. The best thing to keep in mind is that if somebody doesn't know how to play bridge, or poker, or gin rummy, or whatever, the odds are he'll never run your program in the first place, whereas anybody who does run that program will already have an interest in the game. So what you need in this sort of introduction is simply to present the pertinent information about your computer version of the game to the knowledgeable player. Let's look at how this is handled in our poker program.

***Five-Card Draw***

```
60000 TEXT
    : HOME
    : PRINT SPC( 14)"DRAW POKER"
60010 PRINT
    : PRINT
    : PRINT
    : PRINT
60020 PRINT "  COPYRIGHT (C) 1984 BY JIM ME
      NICK"
60030 FOR PAUSE = 1 TO 5000
    : NEXT
60040 HOME
60050 PRINT "THESE ARE THE HOUSE RULES:"
    : PRINT
60060 PRINT "   EACH PLAYER BEGINS WITH $10
      0"
    : PRINT
60070 PRINT "   1$ ANTE"
    : PRINT
60080 PRINT "   JACKS OR BETTER TO OPEN"
    : PRINT
60090 PRINT "   THREE RAISE LIMIT"
    : PRINT
60100 PRINT "   5$ BETTING LIMIT"
```

Line 60000 does a number of things worth mentioning. First, the command TEXT clears away any text window limitations that might be hanging on from whatever came before (running a program does not automatically default everything in your computer. If you were running a program incorporating graphics and then decided to run "Five-Card Draw," you would still be in the graphics mode unless you deliberately removed it. Similarly, if you were running an adventure with a text window such as the one used in "Space Derelict!" this window too might still be in effect if you didn't

otherwise remove it. TEXT takes care of this problem for us, combined with HOME, which gets us to the top of the screen. It's a good idea to begin any program with this TEXT/HOME combo.

The SPC command is one I tend to use rarely, because I usually find it easier simply to type in the missing spaces myself on the program line, visualizing the screen as I go along. SPC(14) in line 60000 means that the printing begins 14 spaces in from the cursor position, which in this case happens to be the extreme left. If the cursor already happened to be in five spaces in from the left, for instance, then SPC(14) would have begun printing 14 spaces away from space number 5. This is somewhat different from the TAB command, in that TAB(X) places the cursor in the Xth column from the left, erasing everything else (if anything) preceding it on the line. For example, if we had printed the word POKER on the far left of a line and specified TAB(14) on that same line, the word POKER would be erased and our next printing would begin in the 14th column, following 13 blank spaces. However, in the case at hand, the SPC and TAB functions are interchangeable, except that we would have had to say TAB(15) to get us into the 15th column (which is where we end up with SPC(14), the value of 1, where the cursor already was, plus 14). The use of SPC is the same whether you're using 40 or 80 columns.

Line 60020 is where you get to exercise your designer's ego. Keep in mind that just because you say a program is copyrighted doesn't make it copyrighted. To make it official you must register your program for a nominal fee with the copyright office in Washington, D.C. Which brings us to a few words about getting your programs published.

I read somewhere recently that writing the Great American Program has lately replaced the quest of writing the Great American Novel. Everybody's writing the former nowadays, and everybody wants to get them published. But how do you protect yourself from unscrupulous publishers who will steal your idea and never give you credit for it? Well, one thing is not to send your program to unscrupulous publishers in the first place (even if it's copyrighted, nothing prevents the unscrupulous publisher from stealing it anyhow). If you really have a program you think is salable, the first thing to do is contact the major software publishers and find out if they even consider submissions from outsiders. If so, just send them the program and don't worry about it. These publishers have better things to do than steal your ideas. First, if your idea is no good, they won't want to steal it, and second, if it is good, it's easier for them to publish it than to steal it. They're not in the business of ripping off newcomers. These are honest businesspeople looking to make money, and in many cases they also carry a bit of the pioneer spirit with them, a spirit they are eager to share. Of course, you should be a bit wary of a software publisher you've never heard of before who advertises in "Bum Biters Monthly," but other than that you can feel reasonably confident that well-known publishers are operating on the up and up. If a publisher decides to take on your program, then it's time to worry about copyright.

But back to introductions. As you can see, the "Five-Card Draw" one is brief and to the point. Knowledge of the game of poker is assumed, so the only things you get here are the special points pertinent to the computer version. First, we tell the player that everyone has $100. It is a good idea for

you as designer to set the bankroll limitations rather than allowing the player to enter his own bankroll. This puts some preplanned boundary to the game. If the player has an infinite supply of money, or close to it, he loses some of the fun of trying to manage the funds at hand. In this game, $100 goes quite a long way. If you want to make the game a little more cutthroat, change this amount down to $50.

In this program the anteing is handled automatically, which is why I've explained in the introduction how much it is. It could just as easily be shown later on the screen in a statement such as:

PLEASE ANTE $1

A fielder's choice, so to speak.

Jacks or better is the name of this game, according to line 60080. As I mentioned earlier, it was a lot easier to program draw poker this way than with anything opens. And, as a player, I prefer the traditional jacks or better. The only problem we get in the program as a result is that every now and then we get hands where nobody opens, which means sitting at the computer for a while with nothing much happening while the cards are reshuffled and redealt. Unfortunate, but an occupational hazard, just as frustrating in the real game as it is here.

The three-raise limit is not unusual in the friendly Friday-night poker game, but it serves a special purpose here. If we had put no raise limit on the game, our subroutine that handles the betting would have been all that much more complicated: How would the computer figure out when to stop raising? With the three-raise limit, the computer just stops when we hit the third raise. And finally, the introduction specifies the $5 betting limit, which serves two functions. Again, having such a limit helps us in programming the computers betting moves. Without it, how would the computer know when to stop? In effect, this limitation helps give the computer "players" their version of the poker face. The second aspect of the $5 limit is its relationship to the total bankroll of $100: Even the greediest player will hang around for a while at these prices. We don't want the player to go bust too soon, and too high a betting limit might help him to do just that.

All of which brings us back to random numbers, which started this whole discussion. Witness the rest of the introduction subroutine:

***Five-Card Draw***

```
60110 PRINT
    : PRINT
    : PRINT
    : PRINT "HIT RETURN TO CONTINUE"
60120 POKE 49168,0
60130 X = RND (1)
60140 IF PEEK ( - 16384) < > 141 THEN 60130
60150 RETURN
```

Oh, no, we're POKEing and PEEKing again. Sorry about that, but these are two of the best, with a myriad of uses just waiting to be discovered.

Did you ever wonder how the keyboard communicates with the microprocessor? Well, it's all very simple. The keyboard is one complete unit con-

nected to a mysterious little piece of machinery that collects keyboard input, which in turn is connected to the CPU—the computer itself. The microprocessor, depending on what instructions you give it, interacts with that little piece of collection machinery in various ways. Usually that little machine collects a number of keyboard strokes and just sits on them. When you eventually hit Return, the microprocessor takes and does something with those collected keystrokes. What we're going to do at this point in the program is manipulate that interaction of little machine and microprocessor to our advantage. The first thing we do is print to the screen the message in line 60110, telling the player what is expected of him. Then, in line 60120, we are going to empty that little machine to make sure there are no malingerers in there waiting to screw us up. POKE 49168,0 puts a value of 0 in machine address 49168; i.e., it empties that little machine. We are now starting with a clean slate.

In 60130 we pull a random number using the RND(1) function. The RND command is a three-headed beast, depending on the value we put into the parentheses. A positive number between 1 and 255 (*any* positive number, it's all the same to the computer) pulls off the next random number on the stack generated by the computer. As we discussed before, when you turn on your machine, the same stack of random numbers is always waiting, which is why we're going through all this rigamarole in the first place. Now, if we were to put a 0 in those parentheses—RND(0)—we would always get the same random number we got the previous time. Why would you want to repeat the same random number is beyond me, but that is what a 0 will do. Finally, if you put in a negative number, you will repeat the last *sequence* of random numbers. There may be some value to this, but what it is I couldn't say. For our purposes, we'll always use RND(1), getting a hot, fresh new random number, which is what we're looking for in our random universe.

So now we've cleared the little keyboard collecting machine and pulled out a random number. Then what? Line 60140 is the other half of the equation begun in line 60120. PEEK(-16384) looks inside the little machine and sees what's in there. The value of 141 represents the machine code number for a Return (i.e., CHR$(13) is a Return, and 13 + 128 = 141—the Apple use 128 ASCII characters two times for a total of 256; either trust me or start reading the footnotes in your manuals, because this is really far-out stuff). So this PEEK looks inside and sees if there's a Return in there. If not, processing goes back to 60130, and another random number is generated. Then back to 60140 and the same PEEK, and so on until the player finally presses Return, at which point processing continues on the next line, 60150, the end of the subroutine. The point is, now the collection of random numbers is entirely dependent on how quickly the player hits Return. While the computer is waiting for that thump on the keyboard, it is continuously running through this loop of random numbers as fast as its little legs can carry it, which is pretty damned fast. And we've now conquered the problem of getting a really random number, entirely dependent on the outside condition of how long it takes for the player to hit Return.

As you can imagine, we could have performed a variety of activities other than seeding a random number between lines 60120 and 60140, if we were so inclined. In a way, this function mirrors the GET statement, but with one decisive difference. GET stops dead in its tracks until something is entered

from the keyboard, while the use of this POKE/PEEK team allows us to do whatever we want in the interim. We could have entered a graphics display, for instance, or done some other processing business—you name it. A very convenient and useful team, this one, definitely worth keeping in mind.

If you're really curious and you don't already know, you may be wondering where in tarnation the negative machine addresses are—i.e., why PEEK(-16384). In fact, all machine addresses can be expressed as either a positive or a negative decimal number. If an address is positive, subtract 65536 to get its negative equivalent; if it's negative, add 65536 to get its positive equivalent. In our team above, 49168 = -16368, and -16384 = 49152. This seeming irony is the result of the computer speaking in binary rather than decimal numbers: There are no negative binary computer numbers per se. But I'll save you the course in binary arithmetic, since beyond knowing how to express addresses in both positive and negative decimal numbers, it's entirely useless in our understanding of Basic.

## ARRAYS MADE COMPLICATED

In our adventure game we had very few variables, and it was never very difficult keeping track of what was what and what was meant by it. Unfortunately, that will not be the case in the poker program, as it probably will not be the case in most strategy games you design. The problem is that you have to keep track of an awful lot of information, each piece of which is independent of the other pieces, so you're forced to bring into play a lot of different variables. But one lesson we have already learned from "Space Derelict!" is to use arrays when the situation is ripe for them. If we have variables thematically related to other variables, bringing arrays into play usually makes sense. In "Space Derelict!" we used D(X) for doors, CO(X) for movable objects, and so forth. For the sake of simplicity, you should try to follow through on this practice.

Another worthy practice is always to use the same counter variables—i.e., always use N as in FOR N = 1 TO 10 rather than using N this time, X that time, and Z the next time. For any counter variable like this where once you use it, it doesn't matter what its value is, you might as well use the same one over and over. The smaller the number of variables, the more memory available to your program.

Not counting arrays, there are about 50 variables in the poker program. Some of these variables come up often during the execution, while others have very specific uses in some of the darker corners. As a rule I've tried to make the variables somehow meaningful to me so I could remember what they were all about, so there's one variable called DOUBLECHECK$(X), another called DISCARD(X), and so forth, where obviously I've gone on longer than the two letters that Applesoft can understand. But the point in these cases was not that Applesoft could understand it but that I could; the understanding by the computer was implicit in the operation. If you can keep your variables down to one or two letters, so much the better, but it really doesn't take up much more space to throw in whole words like this, and as a program grows in complexity you might find yourself doing it just as I did. Just remember that only the first two letters count and that DINGBAT and

DIMWIT would be one and the same to the computer when it starts handing out the paychecks.

As with an adventure, our variable initialization/housekeeping takes place early in our strategy programs, but since this is a one-time access we can put it way up in the stratosphere of line numbers. As you will see, all of my 50 variables are not accounted for in my housekeeping, even though, as you'll find out later, some of them are arrays. As we mentioned earlier, if an array has 11 or fewer elements (e.g., D(10) or less—don't forget the 0 value), Applesoft will automatically set aside this space for us the first time the undimensioned array variable is encountered in the program. As for the nonarray variables, there seemed to be no point in noting these here if no initial value was necessary for them; they'll come up soon enough in the ordinary course of events. Again, I'm trying to save myself programming time—why throw in a bunch of REMs when I don't really have to. This time-saving may be all right insofar as the straight variables are concerned, but there is a trade-off in not dimensioning the 11-or-less arrays. The thing is, even if I don't use all of my default allowance of 11, the computer has nonetheless set aside that much space for them. And variables (especially of the string persuasion) take up space in memory, whether we use them or not; the fewer used, the more space you have for more important things. And don't kid yourself into thinking 64K is more than enough for everything you want to do, because it isn't. Sloppiness on a big program can easily cause you to run out of memory, which is why we'll be talking a lot in this part of the book about condensation and space-saving, as well as time-saving in execution. But in this particular instance the actual number of variables set aside but not used is negligible, so we don't have to worry about it.

Rather than bombard you with all my variables right off the top, we'll discuss most of them when we encounter them in the programming. So now let's look at the housekeeping for "Five-Card Draw":

***Five-Card Draw***

```
50000 REM  SET DIMS AND VARIABLE NAMES
50020 FOR X = 1 TO 5
    : R(X) = 100
    : NEXT
50030 REM   DIM P(5) = PLAYER NUMBER - 0/I
        N, 1/OUT, 10/OUT FROM THIS HAND ON
        LY
50040 D = 1
50055 G$ = ""
50060 P$(1) = "MARVIN"
50070 P$(2) = "GERRY"
50080 P$(3) = "WALTER"
50090 P$(4) = "BETSY"
50100 P$(5) = "YOU"
50110 DIM C(40),C$(40),S(40),S$(40),CD$(40
        )
50130 DIM V(5,5),V$(5,5),ST(5,5),ST$(5,5)
50140 DIM CH(52)
50170 DIM B$(12)
50180 B$(0) = "XXXXXXX"
```

```
50190 B$(1) = "I'LL OPEN FOR $"
50200 B$(2) = "I'LL SEE THAT"
50210 B$(3) = "I'LL RAISE YOU $"
50220 B$(4) = "THE BET IS $"
50230 B$(5) = "THE POT IS $"
50240 B$(6) = "YOU HAVE $"
50250 B$(7) = ": OPENER"
50260 B$(8) = "I CAN'T OPEN"
50270 B$(9) = "I'M OUT!"
50280 B$(10) = "I'LL TAKE "
50290 B$(11) = "I'VE GOT ZILCH!"
50300 B$(12) = "                    "
50320 RETURN
```

You'll notice that the line numberings are not necessarily in the usual increments of 10. This is the result of last-minute changes (usually deletions where the information in that missing line has been merged into another line). I could have gone back and renumbered all of these, but I decided to leave them in case you wanted to figure out for yourself what sort of compression was actually done. It's easy to see what happened in 50110–5013:I ended up using commas to concatenate these dimensions.

Line 50020 introduces our first variable, R(X). In this case X has five values, from 1 to 5, each of those values representing one of our five players. R(X) refers to the amount of money each player has (the derivation of R, in this case, from bankRoll—I always use R as my money variable)—$100 each across the board.

Line 50030 is a REM I kept because it's extremely important. The variable P(X) refers to each of the five players in the game from P(1) to P(5). The value of P(X) tells us whether that player is in the game (a value of 0), out of the game entirely (a value of 1), or merely dropped out from this hand only (a value of 10). This is how we can keep track of whether to deal a given player a hand, whether he should bet, etc. You'll find:
IF P(X) < > 0
or variations thereof sprinkled liberally through the rest of the subroutines as a check to make sure our particular player is still with us.

Line 50040 is yet another variable, in this case D as in Dealer. We're initializing this variable with the value of 1 to indicate that player number 1 (or P(D)—P(1)) will deal the first hand. Altogether we will have three variables keeping track of who deals, who bets, and so forth, but D is the granddaddy of them all on which all the rest are based. We'll get to the others when we need them.

As in "Space Derelict!" G$ equals Ctrl-G, the little bell. In "Five-Card Draw" this noisemaker is used much more liberally than in our adventure, primarily because the screen formatting is so complex as to necessitate an additional cue to the player above and beyond the visual that something is happening. You'll see what I mean when we get into the next chapter. You could delete this line if you prefer silent playing. (It will also speed things up a little bit, sometimes with the dizzying effect of watching a movie at double speed. Try it and see what I mean.)

In 50060–50100 you get to meet the opposition, as well as yourself. As you've probably surmised, there is a one-to-one correspondence between

P(X) and P$(X). When we're talking about P(1) we're talking about Marvin; when we're talking about P(2) it's Gerry, and so forth. With the exception of YOU, these players are all handled by the computer, and as promised, each is handled differently. In fact, all four of them are based on my real-life poker buddies, and to some degree the way the computer plays their hands is similar to the way they play in real life. They were my inspiration in coming up with the algorithms for the computer strategies.

In line 50110 we've got our first set of arrays, each of which has been dimensioned at 40. These five variables are the ones we're going to use to represent the cards at play in the game. The reason there are 40 rather than 52 is that in five-card draw with five players you need a maximum of only 40 cards (25 on the deal, plus up to three for each player for the draw). Here we're saving 60 (5 variables × 12 values) variable space allotments by making sure we're not asking for more space than we need. In many card games you will similarly not use an entire deck at once, and this figuring ahead of time can save you memory space (and shuffling time) later on.

C(X) refers to the value of the card, which will range anywhere from 1 to 14, low ace to high ace. C$(X) is the corresponding name for that value, from the numbers 2, 3, etc., up to ace. S(X) refers to the suit; 1 equals clubs; 2, diamonds; 3, hearts; and 4, spades. S$(X) is the equivalent spelling out of the suit name. CD$(X) is the concatentation of C$(X) + S$(X); if C(4) = 11 and S(4) = 3, then C$(4) = "JACK" and S$(4) = " HEART" and CD$(4) = "JACK HEART". All very elementary, and now you can see how we establish both values and names for each of the cards that will be dealt in the hand.

Next we get to the hard part, line 50130. We talked all about arrays in Part One of this book but mentioned in passing that there was more to come of a more difficult nature later on. Well, here it is, the double-decker dimension, and in fact it's not all that complicated to understand, but it can be complicated to manipulate. Let's talk about theory first.

You can have up to 88 dimensions to an array (as in PT(1,2,2,1, etc.), up to 88 numbers separated by commas)—or at least that's what they say in the manuals, always with the caveat that the number 88 depends on various other factors too complicated to enumerate. I'll bet no one has ever used that many dimensions in an array in the history of computing. It's just too complicated to keep track of. But smaller dimensions of two or three elements are not at all uncommon, and their use can be very valuable in keeping track of certain kinds of data.

The best example I can think of to show how the double-decker dimension works is the one used in "Five-Card Draw." In this game we have five players, and each player has five cards. So imagine that player 1 gets his card 1, player 2 then gets his card 1, and so forth up to player 5 getting his card 1. Then player 1 gets his card 2, and so forth. Five players, each with a different card 1, card 2, card 3, card 4 and card 5. Our double-decker array holds that information for us one piece at a time: V(1,1) is the value (same meaning as C(X) above) of player 1's card 1. V(1,2) is his second card, and so on. V(2,1) is player 2's card 1, V(5,4) is player 5's card 4, and so on. V$ is the equivalent of C$, ST the equivalent of S, and ST$ the equivalent of S$.

But what is the reason for all this? In "Five-Card Draw," the double-

deckers allow us to use the same subroutines over and over again regardless of which player we happen to be talking about. In these routines we're interested in the cards a player has from card 1 to card 5, and we'll be evaluating these cards and creating new variables that tell us what a player is holding *in which cards*. For instance, not only will we want to know that a player has been dealt a pair, but also we'll want to know which two (e.g., first and fifth) of the five cards the pair happens to be. Use of the double-deckers allows us to do this.

If all of this sounds hopelessly vague and confusing now, hang on a bit until you see it in action. As long as you can understand that V(1,3) means player 1, card 3, at least you understand the concept underlying multiple dimensions. Their use will become clear later on.

Line 50140 dimensions a counter variable we'll be using when we shuffle the cards. CH(X) as in CHeck will help us keep from drawing the same cards more than once. Later CH(X) will do additional duty, checking for us the individual value of each player's hand.

The final thing worth noting in this subroutine is B$(0). These seven X's will be used in our screen formatting to represent the cards dealt to players 1 through 4 that they can see but the human player can't—in other words, the backs of the cards. The rest of these lines should be self-explanatory, and you'll see them in use soon enough.

## SHUFFLING THE CARDS—MAKE-BELIEVE SPEED

If you pay attention to what's happening in the microprocessor field, you are probably aware of manufacturers' never-ending quest for more speed. On the most dramatic front, advances in processing speed are gained simply by changing the nature of the processor from an 8-bit (such as the Apple) to a 16-bit, a 32-bit, and so forth. Obviously, the more numbers the computer can bite off in one piece, the more work that can be done faster. Speed enhancers of other natures are available as well. You can buy all nature of souped-up DOS programs that work roughly the way your original Apple DOS works, but more quickly. This means that you might have to wait for only 10 seconds to load a program from a disk when previously you might have waited up to a minute.

And in our discussion of structured programming we mentioned how more careful programming, which takes into account how Basic works in our machines, can make our programs run faster. But although most of the time this concern with processing speed seems superfluous, given the already incredible speed with which any computer operates, sometimes it is a question of vital importance. Sometimes you are going to create an algorithm to perform a certain chore, and although that algorithm is a perfectly good one, it will be too slow to be satisfactory. Let's face it: Given the usual swiftness of data-shuffling that faces us every time we sit down at the computer, it can be very surprising and even frustrating when we're just sitting there waiting for the computer to finish what it's doing. Getting around this problem is what this section of this book is all about.

The first thing to do if you design one of these albatross routines into one of your programs is to make sure it's written as "structured" as you can get it. Are there too many GOTOs in it? Is it at the end of the program when it

should be at the beginning? Can it be trimmed down by merging lines, deleting REMs, etc.? After checking all these possible tardiness-makers, if you've still got a slow-moving albatross, perhaps you should try to rethink your algorithm. Would there have been a better way to accomplish the chore that's been slowing you down? Completely throw out that part of the program and see if you can rethink it. Perhaps a speedier solution is just waiting for you to clear your head of your first attempt. Any creative process has its temporary dead ends, and maybe your albatross happens to be one of them. A good library of programs might be helpful to you as well: In a book or a magazine maybe you can find a similar problem, the solution to which you can adapt to your own needs. All these things are worth trying, but there is a last resort when all else fails and you're stuck with your albatross as is.

Despite the fact that I have programmed a number of games incorporating the shuffling of cards, it was only after this book was being typset that I designed a formula for this that I find truly satisfactory. The one or two Basic solutions to this problem that I'd tried to glean from other available programs didn't seem to be any improvement over my own albatross. The problem in shuffling is a simple one: It's easy to generate 52 random cards, but it's not so easy to make sure you're not generating the same ones more than once. Fifty-two *different* cards are needed, and therein lies the rub. Every time the computer generates a new card it has to go back and make sure that card hasn't already been generated; if it has, then the computer has to try again. And as each card is dealt, the harder this becomes, because the computer has to go back over more and more previously dealt cards each time to make sure the new one isn't a duplication. This is time-consuming, even for a computer working at breakneck speeds. And it's time when our player sits in front of the computer and nothing seems to be happening. Not at all a desirable situation.

My penultimate shuffling routine took approximately 15 seconds. By rethinking it, I managed to get it down to about 5 seconds, a substantial savings. But 5 seconds is still 5 seconds, an interval that seems ponderously long for the player with nothing better to do than stare at the blank screen of the computer.

But there is a way around this problem. After we've got our albatross as fast as we can make it, we simply bury it. We integrate it into the program so that even though it takes the same amount of time to process, the player isn't conscious of that time elapsing. While the computer is tirelessly toiling away at this one chore, we mix in another, superficial activity on top of it so that the player still has something to keep him busy and is never the wiser about the albatross that otherwise would have been staring him in the face.

***Five-Card Draw***

```
6000  REM  SHUFFLE
6010  HOME
    : N = - 1
6020  PRINT
    : INVERSE
    : FLASH
    : PRINT "SHUFFLING"
    : NORMAL
    : PRINT
```

```
6060  FOR X = 1 TO 40
6070  C(X) = INT ( RND (1) * 52) + 1
6080  IF CH(C(X)) = 1 THEN 6070
6090  CH(C(X)) = 1
6130  S(X) = INT ((C(X) - 1) / 13) + 1
6140  IF S(X) = 1 THEN S$(X) = " CLUB"
6150  IF S(X) = 2 THEN S$(X) = " DIAMOND"
6160  IF S(X) = 3 THEN S$(X) = " HEART"
6170  IF S(X) = 4 THEN S$(X) = " SPADE"
6180  IF C(X) > 13 THEN C(X) = C(X) - 13
    : GOTO 6180
6190  IF C(X) < 11 THEN C$(X) = STR$ (C(X)
      )
6200  IF C(X) = 11 THEN C$(X) = "JACK"
6210  IF C(X) = 12 THEN C$(X) = "QUEEN"
6220  IF C(X) = 13 THEN C$(X) = "KING"
6230  IF C(X) = 1 THEN C$(X) = "ACE"
    : C(X) = 14
6240  CD$(X) = C$(X) + S$(X)
6245  IF X / 6 = INT (X / 6) THEN GOSUB 62
      70
6250  NEXT
6255  FOR X = 1 TO 52
    : CH(X) = 0
    : NEXT
6260  RETURN
6270  REM  ANTE
6275  IF N = - 1 THEN PRINT
    : PRINT "PLEASE ANTE:"
    : PRINT
    : PRINT
    : N = 0
    : RETURN
6280  N = N + 1
    : IF N = 5 THEN 6310
6290  IF P(N) < > 0 THEN PRINT P$(N) + " I
      S BUSTED."
    : PRINT G$
    : PRINT
    : RETURN
6300  R(N) = R(N) - 1
    : PRINT P$(N) + " IS IN."
    : PT = PT + 1
    : PRINT G$
    : PRINT
    : RETURN
6310  POKE 49168,0
6320  INPUT "ARE YOU IN?";C$
6330  IF C$ = "Y" OR C$ = "YES" THEN R(5)
      = R(5) - 1
    : PT = PT + 1
    : PRINT G$
    : RETURN
6340  IF C$ = "N" OR C$ = "NO" THEN END
6350  GOTO 6320
```

The substance of the shuffling subroutine is contained in lines 6060–6090. In 6060 we set up a loop to get us the 40 cards we'll be needing. In 6070 we generate a random number from 1 to 52. We've already talked about the parenthetical (1) after RND. Multiplying RND(1) by 52 gives us the number of possibilities we want, and the INT function turns those possibilities into integers. The reason we add 1 to the result is that the INT function rounds down the numbers, so without this +1 added on we would end up with numbers from 0 to 51.

In line 6080 we encounter our CH(X) variable, which we originally DIM'd at 52 values. At this point, since no other value has been assigned to them, all 52 CH(X)s equal 0. So the first time through, regardless of the value of C(X), CH(C(X))=0. Processing therefore goes to 6090, where CH(C(X)) is set at a value of 1. We're trying here to make sure we don't get a number from 1 to 52 we already have, so the next time we generate a random C(X) at 6070, if it happens to be one we've already generated, i.e., one which already has a CH value of 1, we go back to 6070 to generate another number until we have one that isn't already taken. After we've successfully generated a random number from 1 to 52 that hasn't already been generated, we simply go on and generate the next one until we have all 40 we need.

Lines 6130–6230 translate the number from 1 to 52 into a card value. Line 6130 gives us S(X), the suit value of our card, in a deceptively complicated equation. C(X)-1 will give us a number from 0 to 51; the INT value of that number divided by 13 will give us a number from 0 to 3, and the +1 gives us an S(X) anywhere from 1 to 4. If you don't understand how this formula works theoretically, try plugging some values into C(X); you should get the hang of it that way. The point is, this gives us just the right suit values of our 52 cards, based on whether the given card falls into the first batch of 13, the second batch, the third batch, or the fourth batch. Lines 6140–6170 are self-explanatory save that we leave a space at the beginning of each string so that CD$(X) will print out 8 HEART rather than 8HEART.

Line 6180 is another cute formula, this time to give us our true C(X) value. What we're doing here is decrementing C(X) by 13 if C(X) is greater than 13, and doing it again and again if necessary until C(X) is not greater than 13. This ultimately gives us a value of C(X) from 1 to 13; line 6230 will take care of the problem of C(X)=1 (later in the program, when figuring straights, we'll be switching this around again). So now we have C(X) values from 2 to 14, and by the time we finished looping through all this 40 times we have 40 different cards and we're ready to deal them out.

Next we come to line 6245, which is how we mask the slowness of this shuffling process. What we've done here is put in a counter that says that if X divided by 6 is an integer, a whole number rather than a whole number and a fraction, then we'll GOSUB to 6270. Looking down to these lines, you'll see now why we set N=−1 in 6010. This subsubroutine simply collects the antes from each player. If N=−1 (our starting point), then 6275 prints PLEASE ANTE on the screen, increments N to 0, and then RETURNs. Six X's later in the original loop we'll come back to 6270 again, and this time we'll skip past 6275 to 6280, incrementing N again, following the procedures in 6290–6300 for values of N less than 5, and 6310–6350 for when N does equal 5. You'll remember that P(N) refers to our player

numbers from 1 to 5. If P(N)<>0, then that player is out of the game, hence the message delivered in 6290. If the player is still in the game, then his bankroll (R(N)) is decremented, the pot (PT) is incremented, and the appropriate message in 6300 is sent to the screen.

If N = 5, then we're talking about the human player, P(5), so we go down to 6310. Once again we clear the little keyboard machine; otherwise line 6320 will print twice because the last Return we pressed to get here will now register as the input C$ in 6320, which we're cowcatching in 6350 to return to 6320 if it's not a "YES" or a "NO"—an odd little problem that you ought to be careful about when you use those particular POKE/PEEKSs. Assuming that our player opts to stay in the game, his bankroll too is decremented and the pot incremented. If he bows out, then we END the program. Although I have not done it myself, you could add a save-game feature not unlike what we used in "Space Derelict!" so that the player could return to the game with each player's bankroll entered into a text file to be pulled out at a later playing. All you have to to do is save R(N) for N equal to 1 to 5 and then throw in a REPLAY SAVED GAME option at the outset of the program to replace line 50020 in the housekeeping section.

What have we done here? By interpolating this sub-subroutine of anteing into our shuffle subroutine, we have completely masked the fact that we've been shuffling the cards. As our player has sat at the computer he's been busy watching each player ante and then anteing himself. As far as the player is concerned, the only thing that separated one ante from the next was a short but suitable pause. But for us, we've won the battle of squeezing a long routine into the program without hindering the playability of the program. This is an important technique that is well worth remembering in your own programming, but it has one significant drawback: It can get very complicated and confusing. In the case of shuffling the cards it was relatively simple because we had an easily adaptable subsubroutine, the anteing, waiting in the wings to volunteer as our decoy. But in many cases you won't have so willing a victim, and you may be setting yourself up for some hard programming work. But it's usually worth it in the long run, especially in a game program. The last thing we do after our shuffling routine is completely looped is reset all our CH(X)s to 0 in line 6255. Now we can RETURN and get into a little poker.

# 10

# THE LOOK OF THE GAME

No matter how good a game you design, if your game doesn't look good, nobody is going to want to play it. In some cases looking good is simply a matter of neatness or clarity; this is especially true of text adventures and nongraphic strategy games such as "Five-Card Draw," where we're doing nothing but manipulating text on the screen, and it's entirely up to us where that text is going to go. In other cases looking good means clever, state-of-the-art graphics, which as I've mentioned a number of times already is not in the purview of this book. But nonetheless it almost goes without saying that the same "Five-Card Draw" game, designed with or without graphics, is going to be a little more appealing in its graphics incarnation, even though both games are otherwise entirely identical. The use of graphics in this instance brings in another whistle and bell, and there's no denying that it's a fairly desirable one. But the creation of these graphics can be very time-consuming, in some cases taking as much extra work as the original program itself. If you want to put in the graphics, be my guest; there are plenty of perfectly good graphics utilities on the market, as well as numerous good and informative articles in the Apple magazines (and if you're a truly hardy soul, there's the original Apple manuals, possibly the worst place of all to look for graphics information). Since most strategy games require fairly simple graphics compared to arcade-type games or even adventures, you can reasonably expect to be able to design a quite decent-looking game. But you'll have put your work in to get it. If you're so inclined, there's more information in Chapter 12 to start you down this road.

Even when you don't use graphics, you still have to work hard to create good-looking screen output, and that is what this chapter is all about. The techniques we will be discussing here are applicable to any kind of program, and although they take some getting used to, they are quite elementary and

easy to operate once you get the hang of them. Mastery of the handful of commands in this chapter can turn you into a true visual whiz—textually speaking.

## 40 VS. 80

Once upon a time, there wasn't much of an issue between an 80-column and a 40-column format on the Apple video screen. Although it was (and is) possible to obtain hardware to transform your II+ from a 40 to an 80, there didn't seem much point to it unless you were in business and willing to shell out a fair amount of money to run more sophisticated word processors and spreadsheets at their optimal levels. For the hobbyist, to whom VisiCalc is about as useful as a dead monkey, why bother? Games were all written for 40, as were most word processing programs, so what did it matter?

But the introduction of the IIe changed all that, not because the hobbyists suddenly decided that 80 was more desirable—a moot point—but because retailers were starting to throw in the 80-column adaptation (the IIe is still basically a 40-column machine) as almost a given in their sales packages. And let's face it, for the now minimal extra investment, if you are going to do any word processing or other use where the 80-column enhancement might be desirable, why not go with it? And for a little more investment there's the 128K version. I can see the attraction.

But despite all of this, we're going to be concentrating on 40 columns only in this book, because that's still what most people have, and the technical differences between the two are small enough so that if you understand 40-column formatting thoroughly, 80-column formatting requires only a tiny bit of extra work. Also, in many cases 40 column makes sense, especially in the games we've been designing in this book. The whole point of 80 columns is to squeeze more information onto the videoscreen, but if you have no need of a lot of information on the screen, why use it? It's very simple for 80-column users to switch down to 40 columns if the situation calls for it. That is up to you. The point is for you to decide in advance of your design whether you think the 40 or the 80 would be better for the game you're working on, and then go with the better choice. And don't get talked into upgrading your non-IIe Apple into an 80 just because it's "state of the art." Don't buy peripherals you don't need; you have enough to worry about buying all the little goodies you think you *do* need.

## IN AND OUT THE WINDOW

We've already touched on one element of screen formatting in "Space Derelict!"—i.e., the text window—although we didn't discuss the concept in too much detail. We simply used it for our own needs and let the devil take the hindmost. But we should go through it again here, throwing in the few other important window-formatting commands that were of no use to us in our adventure program.

The point of setting a different text window from our normal default of 40 (or 80) by 24 is to allow us to keep some portions of the screen static while

other portions keep moving. In our adventure game, this meant keeping the room description steady at the top of the screen while our conversation with the computer scrolled below. In a situation similar to the adventure, where you have text on the top or bottom or on either side that you want to hold steady, the text window is the way to go. Let's go over all the window commands.

POKE 34,X was how we set the top line of our window. The X was any value from 0 to 23 (a lot of computer counting starts with 0), so if we set POKE 34,5, for instance, we would end up with 6 lines of unmovable data at the top and a regular scrolling of data below. POKE 35,X was the other side of the coin, a window set from the bottom, similarly from 0 to 23, with 23 again being the bottom line of the screen. The default value of address 34 is 0, while the default value of address 35 is 23. If we wanted to remove these windows, we'd simply POKE these default values back into those addresses. Note also that the TEXT command resets the window back to its original default parameters in all four directions at once (top, bottom, left, and right).

POKE 32,X sets the left edge of the window. X can have a value of 0 through 39 (or 79, with the 80-column card), 0 being the default. POKE 33,X goes from right to left, with values from 1 to 40 (or 80)—you tell me why left starts at 0 while right ends at 1; the manuals promise a nice system crash if you try to POKE 33,0.

As you can imagine, the problem with resetting the text window is not only the complications that can arise in so doing in a program in progress (remember how careful we had to be in "Space Derelict!" when we were changing a room description?) but also the ultimate limitations such window dressing result in. Although you can get around these limitations with clever (if time-consuming) HTABing and VTABing, there's not much point to trying unless you definitely need that mix of static and movable information, as in the adventure. In most cases, similarly clever (and not so time-consuming) use of HTAB and VTAB *without* the text window will usually do the job perfectly well.

## VTAB, HTAB

I'll say this once, and I won't say it again. In 80-column format, substitute POKE 36,X for HTAB values over 40. The legal command POKE 36,40 would equal HTAB 41, an illegal command, and this holds true up to legal POKE 36,79 which equals illegal HTAB 80; the discrepancy in the two numbers is that the POKE starts at 0 and HTAB values start at 1. (Did you ever think that sometimes the computer manufacturers were out to get you?) After any PRINTing using POKE 36 as an HTAB, remember to POKE it back to 0 again for normal PRINTing. End digression, and now let's discuss HTAB and VTAB for our audience at home.

Unless we do something to stop it, every time we send a PRINT statement from the computer to the screen, that PRINTing begins at the very left of the screen, right below the last line that was printed. If a HOME command has most recently been executed, then the PRINTing begins at the top left of the screen; if text already extends down to somewhere in the middle of the

screen, then the PRINTing begins to the left directly below what's already there; if the screen is full, then the text is scrolled up one line: Our new data appear at the bottom, and every other line moves up one, while the top line disappears altogether. This is all very logical and in many situations quite desirable. But sometimes we don't want data to scroll up on the screen. Maybe we're asking our user to type his name on a blank line on an imaginary form, or showing a game score that we always want to appear at the bottom right corner of the screen, or whatever. This is where VTAB and HTAB come in handy.

In most games there is some sort of playing field that remains static while the pieces keep moving. This concept is obvious in chess or checkers or backgammon, but it's also quite true of "Five-Card Draw" (and just about any other strategy game you'll be designing). In "Five-Card Draw" the field is our five players sitting around a table. Somehow we have to represent this gang of five on the screen; we have to know who they are and where they are sitting in relationship to each other. We have to know who is still in on a given hand and who may have dropped out. We want to know if a player has busted and left the game altogether (in which case we don't want to see him at all). In addition, we want to keep an eye on the pot so we'll know how much is at stake, and we want to be able to see our own stack of chips so we'll know where we stand. We'll also want to see our own hand and at least the backs of the cards of the other players. And finally we'll want to know what the other players are doing, whether opening or passing or raising, and how many cards they are drawing. Needless to say, it's best to design your game first on paper to show exactly where eveything will go on the screen; judicious and frequent use of V&HTAB will allow us to execute it.

Here's how we're going to accomplish all of that in "Five-Card Draw." The screen will be divided into six sections, two columns of three sections each. The top left-hand section will be player number 1, Marvin; on his right will be player number 2, Gerry. Directly below Marvin will be player number 3, Walter, and next to him will be our only female cardsharp, Betsy. The human player will have the section directly below Walter, and to the right of the player will be a box of ongoing information including the amount in the pot, the amount of the present bet, and so forth. Or to put it another way, the screen will look like the layout on the next page.

By "chat" I mean the information about what that player is doing, be it a bet or how many cards he is taking or whatever. On the screen the first line, the words MARVIN and GERRY, will occur at VTAB 1, and every line thereafter is VTAB X in relation to line 1. The middle break, the words GERRY, BETSY, etc., begin at HTAB 20. So the word BETSY, for instance, would be placed at VTAB 9, HTAB 20—and you can figure out the rest for yourself. In the 40-column mode this all looks very nice, perfectly centered, not too busy, and therefore easy to keep track of—all important elements in screen formatting.

VTAB X works from 1 to 24, the length of the screen, and HTAB X works from 1 to 40, the width of the screen. To place the word BETSY where we want it, we would need three commands:

```
VTAB 9
HTAB 20
PRINT "BETSY"
```

```
MARVIN                GERRY
XXXXXXX               XXXXXXX
XXXXXXX               XXXXXXX
XXXXXXX               XXXXXXX
XXXXXXX               XXXXXXX
XXXXXXX               XXXXXXX
chat                  chat
space                 space
WALTER                BETSY
XXXXXXX               XXXXXXX
XXXXXXX               XXXXXXX
XXXXXXX               XXXXXXX
XXXXXXX               XXXXXXX
XXXXXXX               XXXXXXX
chat                  chat
space                 info box:
YOU                   XXXXX
XXXXXXX               XXXXX
XXXXXXX               XXXXX
XXXXXXX               XXXXX
XXXXXXX               XXXXX
XXXXXXX               XXXXX
chat
```

This would place the word BETSY exactly where we wanted it and would have no effect whatever on the rest of the screen. After all the cards are dealt, our "chat" lines are going to be constantly changing, as will the information in our info box, whereas everything else on the screen will, for the most part, stay the same. By using V&HTAB, we can accomplish this miracle.

This is the programming in "Five-Card Draw" that sets up the screen for us. Prior to this section of the program, what we had been looking at on the screen was the anteing process.

***FIVE-CARD DRAW***

```
18000 REM  DEAL
18010 HOME
    : INVERSE
    : GOTO 18050
18040 PRINT B$(0)
    : PRINT G$
    : RETURN
18050 IF P(1) < > 0 THEN 18070
18060 PRINT P$(1)
18070 IF P(2) < > 0 THEN 18090
18080 VTAB 1
    : HTAB 20
    : PRINT P$(2)
```

```
18090 IF P(3) < > 0 THEN 18110
18100 VTAB 9
    : HTAB 1
    : PRINT P$(3)
18110 IF P(4) < > 0 THEN 18130
18120 VTAB 9
    : HTAB 20
    : PRINT P$(4)
18130 VTAB 17
    : HTAB 1
    : PRINT P$(5)
18140 FOR X = 2 TO 14
18150 IF X > 6 AND X < 10 THEN 18220
18160 IF P(1) < > 0 AND X < 7 THEN 18190
18170 IF P(3) < > 0 AND X > 9 THEN 18200
18180 VTAB X
    : HTAB 1
    : GOSUB 18040
18190 IF P(2) < > 0 AND X < 7 THEN 18220
18200 IF P(4) < > 0 AND X > 9 THEN 18220
18210 VTAB X
    : HTAB 20
    : GOSUB 18040
18220 NEXT
18240 FOR X = 18 TO 22
18250 VTAB X
    : HTAB 1
    : PRINT CD$(X + 3) + B$
    : GOSUB 63020
18280 NEXT
18290 NORMAL
    : GOSUB 31000
    : RETURN
```

Line 18010 clears the screen with a HOME and puts us into INVERSE printing. The use of INVERSE gives us a semigraphic look to what we're doing, which in this case is laying down what looks to be cards on the screen. From here we skip to 18050 (we'll come back to 18040 shortly). The first thing we do in each instance in this routine is make sure that the player who is up is still in the game; for a player to be in the game his P(X) will have a value of 0. So 18050 determines whether P(1), Marvin, is still in the game. If so, line 18060 prints his name at the top left-hand corner of the screen (because of the HOME above, we don't need V&H in this instance—printing will default to VTAB 10:HTAB 1). From here to 18130 we replicate the process, putting each player's name in the appropriate place on the screen, provided that player is still in the game. Line 18130 puts YOU, the human player, in position; no test for the value of P(5) is necessary because this condition will have been handled elsewhere. (P(5) not being equal to 0 is a given in this part of the program; there's no point in having the computer playing while the human just sits there and watches—at least at the outset of any given hand. In other instances, when the player drops out in midhand, he will indeed watch his opponents playing without him.)

The actual dealing of the cards begins in line 18140, and in the program-

ming that follows you'll see the continual testing of the value of P(X). We must constantly make sure we're not dealing to a busted player. The X = 2 to 14 condition in 18140 will set the VTAB values for us. Here's how the sequence goes.

If X = 2 our conditionals skip us past 18150 and down to 18160. Assuming P(1) = 0 we go to line 18170, we also skip (because X is not greater than 9). Then 18180 sets the cursor at VTAB X (in this case VTAB 2), HTAB 1, then goes to the nested subroutine at 18040. Line 18040 simply prints "XXXXXXX" (B$(0)) on the screen with a ring of the G$ bell and then Returns. If player 2 is in the game per line 18190, then he will get the next card, courtesy of line 18210. Processing will then pass through to the NEXT at 18220 and will do it again with X equal to 3; and so forth until all the cards are dealt to the extant computer players.

The rest of the routine gives the human player his cards. His VTAB begins at 18, hence line 18240. For simplicity's sake in 18250 we make things a bit complicated. The point is, whereas before we wanted to see the backs of the cards (the seven X's), in this case we want to see their fronts, which would be, in the case of the human player, CD$(21) through CD$(25). To keep our programming efficient we express these values from 21 through 25 in terms of our X, which ranges from 18 to 22. Since the X values we have are three points less than what we're looking for, we simply augment X by 3 in line 18250—i.e., PRINT CD$(X+3). The GOSUB at the end of this line is merely a branch to a pause, to keep things from happening too quickly.

By now you've realized two odd, important things. First, the cards have not been dealt in a "normal" fashion (one to each player in turn). That's okay because of the second odd, important thing. With the exception of the human player, we have ignored entirely the true values of the cards we have been dealing. It has made no difference in the program whether Marvin has been dealt four aces or two 3's or a straight flush in diamonds, insofar as this dealing routine has been concerned. The reason for this is that our actual deal, the assigning of the C(X)'s, S(X)'s, and so forth, has already been handled in our shuffling routine, and all we're really doing here is giving the human player something to look at. As far as the computer is concerned, nothing is needed on the screen for the game to progress in an orderly, well-controlled fashion. That is one of the differences between computers and humans and also one of the reasons why it doesn't matter whether we're talking about text display or graphics. The cards and the game are going to remain the same no matter what takes place on the screen. We've chosen to go with a text display, although we could have used graphics. The game is the same either way and will stand or fall on its merits of playability provided we've given the human something visually feasible he can play with.

Line 18290, before RETURNing to our master routine, which began at line 10000, has one more surprise for us, another nested subroutine at 31000. This is our so-called info box.

## ONGOING INFORMATION

When you sit down to play a hand of "Five-Card Draw" with human opponents, there are a number of givens in the situation that you seldom pay close

conscious attention to. For instance, you don't regularly analyze the exact size of the pot: either there are a lot of chips in it, or there aren't; you know at a glance what sort of money is at stake at any point in the game. Similarly, the size of your own stack of chips gives you a general idea of how you're doing without you're having to add them up every two seconds. In our computer game, however, we don't have these visual reminders, so we have to invent them.

You can be almost certain that just about any strategy game you design will need some area of the screen devoted to an ongoing tally of the game, or at least an updated score after each round of play. The important thing for you to decide is how much ongoing information is necessary to the player, and then where to put it. In "Five-Card Draw" we had the bottom right-hand corner of the screen nicely left over for us, so that's where our information will be going. That's the easy decision; what to put there is a little harder.

First we want to know who the dealer is the deal will pass (of course, from player to player after each hand). We'll also want to know how much money is in the pot and how much money we have ourselves. Since it is not particularly crucial at all times to know how much money the other four players have, this information is left out. If the hand has been opened by one of the players, we'll want a reminder of which player that was. Also, we'll want to know how much the bet is. If the opener bets one dollar and everyone sees that bet, no problem, but if the opener bets 2, the next player raises 3, the bet to you is 5, the next player raises 3, etc., the bet comes around to you again as 3—it's usually not hard to keep track of this in a real game, but the fact that real money is not changing hands here tends to complicate things for us. Hence the constant reminder. And finally, we'll want to keep track of the number of raises because of the three-raise limit.

This is all very simple, and all it requires is that we place the appropriate information into the appropriate V&HTAB locations.

***FIVE-CARD DRAW***

```
31000 REM  SET ONGOING INFORMATION
31010 IF ST < > 0 THEN VTAB 17
    : HTAB 20
    : PRINT P$(ST) + B$(7)
31030 VTAB 18
    : HTAB 20
    : PRINT "DEALER: " + P$(D)
31040 VTAB 19
    : HTAB 20
    : PRINT B$(12)
31050 VTAB 19
    : HTAB 20
    : PRINT B$(4);BE - LB(OP)
31080 VTAB 20
    : HTAB 20
    : PRINT "NUMBER OF RAISES=";NB
31110 VTAB 21
    : HTAB 20
    : PRINT B$(5);PT
```

```
31120 VTAB 22
    : HTAB 20
    : PRINT B$(12)
31130 VTAB 22
    : HTAB 20
    : PRINT B$(6);R(5)
31140 RETURN
```

The only problem with understanding this routine is that it introduces a number of new variables, but we're going to have to digest them sooner or later, so we might as well get it out of the way now.

Line 31010 takes us right to the first one, ST. ST, as it turns out, refers to the player from 1 to 5 who makes the opening bet. So if ST = 1 then player number 1, Marvin, was the opener; if ST = 2 it was Gerry, and so forth. If ST = 0 then nobody has opened yet, so we skip this line entirely. P$(X), you'll remember, was the name of player X, so P$(ST) will give us the name of the opener. B$(7) = ": OPENER".

From this point on you'll occasionally see a clearing of each line (the printing of B$(12)) before the line is printed. This is to make sure we don't leave any old information behind us. For instance, let's say the bet to Walter is $11. He throws in his chips, and then the bet to Betsy is $7. Between Walter's bet and Betsy's we'll have cleared this part of the screen so that we'll print the appropriate $7 when the time comes. What we don't want is the leftover second digit of Walter's bet, turning our $7 into $71 (some of this might make more sense if you look at it on your screen rather than trying to read about it; it's all very clear when it's right there in front of you). It would have been unnecessary to clear the ST line this same way, since that line will always be empty to begin with and will never be replaced during a hand: There's only one opener in any hand of poker.

Line 31030 gives us the dealer (and again we have no need to clear—there's only one dealer per hand). We've already met the D variable, which we originally set at 1 in our housekeeping routine. Line 31050 opens up a whole new can of worms, which I'm going to delay for a while. Suffice it to say the BE-LB(OP) equals the amount of the bet to a given player. This value will be accurate regardless of which player's turn it is. BE-LB(OP) comes from the betting subroutine of the program, which for me proved to be among the most difficult, probably because there were so many things going on at the same time to keep track of. We'll save all that for the next chapter.

Line 31080 tells us the number of raises, which obviously is the variable NB (as in Number of Bet). PT is the pot, and R(5) is the amount of money in the human player's bankroll, which brings us to the end of this subroutine.

Now we've put everything we need on the screen. The player sitting at his computer will see visual representations of five players including himself, as well as a running tally of how he's doing and what's going on in the game. Next we need to set up subroutines for the visual progress of the game itself.

Looking back at our schematic of the screen, you'll find room set aside for "chat." It is on this part of the screen that each player tells the human what he is doing, be it betting, taking cards, dropping out, whatever. Each player, including the human one, has his own chat line. In the case of the human, this is where the computer will be requesting player inputs and, occasionally, sending a message or two back.

We don't have to do much in this subroutine other than clear the appropriate chat line space with a B$(12) and then reposition the cursor at that same space for the printing. The programming looks like this:

***FIVE-CARD DRAW***

```
7000  VTAB 7
    : HTAB 1
    : PRINT B$(12)
    : VTAB 7
    : HTAB 1
    : RETURN
7040  VTAB 7
    : HTAB 20
    : PRINT B$(12)
    : VTAB 7
    : HTAB 20
    : RETURN
7080  VTAB 15
    : HTAB 1
    : PRINT B$(12)
    : VTAB 15
    : HTAB 1
    : RETURN
7120  VTAB 15
    : HTAB 20
    : PRINT B$(12)
    : VTAB 15
    : HTAB 20
    : RETURN
7160  VTAB 23
    : HTAB 1
    : PRINT B$(12)
7180  VTAB 23
    : HTAB 1
    : RETURN
```

All we do is clear the lines and reposition. The routine can be accessed for any individual player however we need it, simply by GOSUBing the correct lines (GOSUB 7160 for the human player, for instance, or GOSUB 7080 for player 3, Walter). The reason that the human player has his access broken down into two lines instead of one is that occasionally the line clearing will not be necessary, so why bother with it? This whole subroutine is placed fairly low in the flow of the program because we use it quite frequently, and there's no point in slowing ourselves down unnecessarily.

Whereas the above routine will be used on a per-player basis whenever we need it, every now and then in the game we'll want to clear everybody's chat lines all at once. For instance, after all the players have completed the first round of betting, each is going to ask for a certain number of cards. We don't want to be looking at a screen where one player says "I'LL SEE THAT" while another player is saying "I'LL TAKE 2 CARDS." That's a sort of visual cacophony similar to all the players at the table talking at once. So subroutine 19000 accesses all the chat sections, clearing them one at a time so the next part of the game can begin.

*FIVE-CARD DRAW*

```
19000 REM  CLEAR SCREEN A BIT FOR TIDINESS
19010 GOSUB 63000
    : GOSUB 63000
    : GOSUB 63000
19020 GOSUB 7000
    : PRINT G$
19030 GOSUB 7040
    : PRINT G$
19040 GOSUB 7080
    : PRINT G$
19050 GOSUB 7120
    : PRINT G$
19060 GOSUB 7160
    : PRINT G$
19070 NB = 0
    : BE = 0
    : FOR I = 1 TO 5
    : LB(I) = 0
    : NEXT
    : GOSUB 31000
    : GOSUB 63000
    : RETURN
```

An easy one, that, clicking off the accesses one by one, starting with Marvin and ending up with the human, ringing the old bell after each one. Even though we haven't said explicitly what some of those variables are in 19070, as you can see, they are all cleared to 0 prior to another access of the 31000 info-box routine. Since this subroutine occurs only at landmark junctions in the play of the game, the information rendered in those variables has become worthless, so we get rid of it. However, we will still see all the rest of the info-box information, with zero values for the information represented by those emptied variables.

As a point of information, I could have squeezed all the data from 19010 through 19060 onto one line, but there comes a point when condensed programming for speed simply becomes obsessive behavior. If I were pressed for space, however, I certainly would have done it.

By the way, you've probably noticed the arithmetic relationship of the lines accessed in this routine: 7000, 7040, 7080, 7120, and 7160. Unfortunately, in Applesoft we cannot use a routine such as:

```
FOR X = 7000 TO 7160 STEP 40
GOSUB X: PRINT G$
NEXT
```

The temptation is strong, but Applesoft does not recognize variables in GOTO and GOSUB statements. Sorry about that.

## DROPPING OUT

There are a number of little sidedish routines in the poker program also related to screen formatting that we won't bother talking about now. They're small and easily digested, tiny variations on what we've been doing up to this point. Aside from these, two more major screen events and one minor one remain to be discussed. The first of these concerns when a player drops out of the game, and the second concerns when the hand is over and all the players show their cards. The third, minor one points to the winner of the hand.

As a hand progresses, it is not unusual for a player to drop out, either because his cards are no good or because the betting has gotten too steep for him. When this happens we want to remove that player's cards from the screen. We most pointedly do not want to show what those cards are if they belong to one of the computer-operated opponents. Just as in real-life poker, when a player drops out, his hand is discarded face down and no one ever finds out what he was holding. On the video screen we want to represent this by deleting either the backs of the cards, in the case of a computer opponent, or the faces of the human's cards, depending on who is dropping out. It is perfectly acceptable for the human to drop out of any hand at any time; the hand will progress without him, just as in real poker, and he will see that hand progress as a spectator. He will rejoin the game at the next deal.

There's nothing complicated about this subroutine, which proceeds from lines 40000–40320.

*FIVE-CARD DRAW*

```
40000 REM  CHANGE DISPLAY WHEN A PLAYER FOL
        DS
40010 ON OP GOTO 40020,40080,40140,40200,40
        260
40020 FOR I = 2 TO 6
    : VTAB I
    : HTAB 1
    : PRINT B$(12) + G$
    : NEXT
40070 RETURN
40080 FOR I = 2 TO 6
    : VTAB I
    : HTAB 20
    : PRINT B$(12) + G$
    : NEXT
40130 RETURN
40140 FOR I = 10 TO 14
    : VTAB I
    : HTAB 1
    : PRINT B$(12) + G$
    : NEXT
```

```
40190 RETURN
40200 FOR I = 10 TO 14
    : VTAB I
    : HTAB 20
    : PRINT B$(12) + G$
    : NEXT
40250 RETURN
40260 FOR I = 18 TO 23
    : VTAB I
    : HTAB 1
    : PRINT B$(12) + G$
    : NEXT
40320 RETURN
```

The value OP in 40010 once again represents any one of our players from 1 to 5 (the exact derivation of OP, a variation of D, need not concern us now). The rest of the lines simply print a B$(12)—our string eraser, so to speak—in the appropriate areas for the appropriate player.

The subroutine for displaying the hands is a tad more complex. At any point in the game, the players are holding these cards:
Player 1—CD$(1 through 5)
Player 2—CD$(6 through 10)
Player 3—CD$(11 through 15)
Player 4—CD$(16 through 20)
Player 5—CD$(21 through 25)
This is quite logical, and if you keep it in mind the hand-display routine will make total sense.

***FIVE-CARD DRAW***

```
47000 REM  SHOW CARDS
47010 IF P(1) < > 0 THEN 47050
47020 FOR Y = 1 TO 5
    : HTAB 1
    : PRINT B$(12)
    : INVERSE
    : VTAB Y + 1
    : HTAB 1
    : PRINT CD$(Y) + G$
    : NORMAL
    : GOSUB 63020
47040 NEXT
47050 IF P(2) < > 0 THEN 47090
47060 FOR Y = 6 TO 10
47070 VTAB Y - 4
    : HTAB 20
    : PRINT B$(12)
    : INVERSE
    : VTAB Y - 4
    : HTAB 20
    : PRINT CD$(Y) + G$
    : NORMAL
    : GOSUB 63020
```

```
47080 NEXT
47090 IF P(3) < > 0 THEN 47130
47100 FOR Y = 11 TO 15
47110 VTAB Y - 1
    : HTAB 1
    : PRINT B$(12)
    : INVERSE
    : VTAB Y - 1
    : HTAB 1
    : PRINT CD$(Y) + G$
    : NORMAL
    : GOSUB 63020
47120 NEXT
47130 IF P(4) < > 0 THEN 47170
47140 FOR Y = 16 TO 20
47150 VTAB Y - 6
    : HTAB 20
    : PRINT B$(12)
    : INVERSE
    : VTAB Y - 6
    : HTAB 20
    : PRINT CD$(Y) + G$
    : NORMAL
    : GOSUB 63020
47160 NEXT
47170 RETURN
```

This routine will start with P(1) and go through each player who is still in the game (i.e., P(X) = 0), displaying each player's cards one at a time. We'll use the counter variable Y to get us through it all, setting up various 5-value FOR . . . NEXT Y loops for each player. What I've done here is use Y as a steady CD$ counter, so you'll see Y slowly progressing from 1 to 20 (we don't have to display the human's already displayed hand) to reach that point in each case where we can say PRINT CD$(Y). This gives us an anchor in the routine, and more importantly, it gives us an anchor variable value through which most of the other values can be expressed.

In 47020, for example, we set Y from 1 to 5. What we want to do is "turn over" each of P(1)'s cards, one at a time, which means we have to go to his appropriate VTAB card lines, which range from VTAB 2 to VTAB 6, erase the inverse "XXXXXXX", and print instead the five CD$(Y)s. To simplify matters, we express the VTAB value in terms of Y—namely, Y + 1 (the relationship of 1-to-5 to 2-to-6). This is similar to what we did in the original deal, lines 18240–18280. To master V&HTAB you have to be capable of such number juggling, which in the long run is much more effective than any possible alternative. After player 1 has laid down his hand, we go on through the rest of the players in the same fashion, RETURNing in 47170.

All the last minor routine does is access the lines on which we've printed the names of the players:

***FIVE-CARD DRAW***

```
46100 VTAB 1
    : HTAB 1
    : RETURN
```

```
46200 VTAB 1
    : HTAB 20
    : RETURN
46300 VTAB 9
    : HTAB 1
    : RETURN
46400 VTAB 9
    : HTAB 20
    : RETURN
46500 VTAB 17
    : HTAB 1
    : RETURN
```

When a player wins we're going to FLASH his name. These minisubroutines get us to the right place.

As I said, this just about wraps up screen formatting, which ultimately boils down to a relatively simple business, as programming goes. The most important thing to keep in mind is to make the screen as attractive as possible. A good look enhances a game; a bad look kicks it in the pants.

# 11

# THE MEAT OF THE GAME

Early in this book we explained every little thing no matter how insignificant. Now there are fewer explanations, on the assumption that you have at this point absorbed a lot of programming lore and can see the logic in complex statements that might have been unclear earlier on. We can pull away now and take a broader look at what we're doing. You've become a Basic literate: We've covered just about every command, and we've been through so many trenches now that you really do know what the battle is all about. You've learned quite a bit (and so have I). This points up one of my most fervent beliefs about programming: The best way to learn it is to do it, and studying other complete, understandable programs is almost as good as creating them yourself. Unfortunately, most good programs are kept under lock and key; the manufacturers copy-protect them, and we never get to see how the state-of-the-art sophisticates are really doing it (although, to be honest, most purchased programs are written in machine language). But we do have the chance in books (sometimes) and in magazines (often) to strip away this cloak of secrecy. After investing the roughly $2000 you plunked down to obtain your Apple, you really should take the next step and get yourself at few $20 magazine subscriptions. The number of magazines devoted solely to the Apple seems to grow daily, and while they all have some similarities (for instance, the same advertisements, which will start to haunt you in your dreams), they also have their own personalities. *Nibble,* for example, one of the most popular Apple-only magazines, consists primarily of industrial-strength complex programs, with the accent on utilities (DOS enhancers and so forth). You won't understand much of *Nibble* for the first year or so, but save those back issues—there's some strong programming in them there pages. *Softalk* is more wide open—more reader-friendly?—than *Nibble* (and is available free to new Apple owners, just to get them hooked into paying

when the freebies run out—a clever strategy). The accent here is more on how-to, with regular columns devoted to different subjects such as DOS, graphics, Basic for beginners, and so forth. Also, there are a lot of articles of general Apple interest on software publishers, significant computer events, famous personages, etc. *Softalk* is my personal favorite. Somewhere in the middle is *InCider,* more intense (and with more copyable programs) than *Softalk* but not quite as heavy-duty as *Nibble*. Then, of course, there are the general magazines such as *Byte, Personal Computing, Creative Computing,* and so forth. These too vary in intensity, and the rule of thumb I follow is to buy them on the newsstands if they have articles of interest to me, which is about 25 percent of the time for any one of them. Here, of course, it all depends on what you're interested in. There is a lot of information out there, and it seems to change all the time; magazines are your best connection with what's what with your Apple.

From here on this book will still have plenty of information, including new tricks on some of the old commands we've already been playing with, but the concentration will be more on structure and design and logic than on the actual programming. If you were to stop reading this book right now, you would already know enough to write some pretty good original programs on your own. But stick with us and you'll learn a little more.

This chapter will include all the rest of the poker program. It will concern itself with the reading of the cards by the computer, the creation of the four computer players, and the process of betting. It will cover both the first round when the cards are originally dealt, plus the second round of drawing new cards and betting some more. Finally, it will show how the winner is decided. Needless to say, that means a hell of a lot of programming and a very long chapter. I had to decide whether to bunch all the similar routines together, or to follow the flow of the game. Because at this point the most important thing to keep in mind is design structure, I have gone with the latter approach, which means we'll be bouncing around a little to different sorts of routines rather than concentrating on the related routines within the program, but by following the flow we'll be able to keep in mind more clearly how the program works overall. Since you now have a good foundation in how Basic works, this is the more logical approach. And first we'll need to get a good idea of the "personalities" of the computer-manipulated players.

## MEET YOUR OPPONENTS

The five-card-draw program in this book pits the human player against four computer-controlled players named Marvin, Gerry, Walter, and Betsy. You'll learn more about these four opponents when you see how they're programmed, but you might wonder right off why these four, and why these four styles of play? It's very simple: These four people really exist—almost. They are, in fact, my real poker buddies, with some slight variations for the sake of the game—most noticeably Betsy, who is in real life named David (I felt that we needed some female presence in this book; sorry, David).

This use of human models made my job a bit easier. And I like to think it was a good idea worth propagating. The real-life models gave me a starting

point and a focus. And now when I play this poker program I can really imagine my friends playing the way their programmed counterparts do (although lately I've been noticing the divergence that I programmed in for the sake of a better computer game). I didn't use my friends 100 percent, but they were certainly the jumping-off point, and some came in closer than others.

As I said, you'll learn more about these four when we get into the game, but I thought I'd introduce them now just to flesh them out a bit. Marvin is the most casual of the four. His philosophy of poker is that it's a fun night out with the boys, and whatever he loses is simply the cost of an evening's entertainment. He also has a philosophy—"That's why they call it gambling"—when he goes for a particularly unlikely draw, such as two cards for an inside straight. People like Marvin make gambling casinos the profitable ventures they are, because Marvin, who is in all other respects a man of awesome intelligence, is a terrible card player. And in our program, Marvin is the easiest opponent to beat, but he's so unpredictable that he's hard to pin down. You'll see why when you sit down against him.

Betsy is one step up from Marvin, by virtue of being somewhat of a student of the game and trying a few things that Marvin would never think of. Betsy takes most of the same unlikely chances as Marvin, but she's a bit more careful. She also firmly believes in bluffing and does so at the drop of a hat, usually at the cost of a lot of chips. The problem is, she can't always be bluffing, and how do you tell if she's really got something?

Walter is on the other side of average. What Walter tends to do that keeps him from hitting the heights is, as they say in the game, betting on the come—staking your money on cards you haven't gotten yet. An instance of this would be making a substantial raise on the first round of betting when you've got a four flush. It's a risky business, this betting on the come. Walter is a more reasonable bluffer than Betsy, and Walter tends to keep a kicker now and then. Walter is probably the sneakiest player of the four.

Gerry is the closest we'll come to a top-drawer player. In the real world my Gerry is not bad, but he's not as good as I've made him here. This Gerry is good at money management: He doesn't bet on the come, he's a conservative bluffer, he knows when and how to draw. The problem is that he plays a fairly dull game, although he does tend to do quite well over the long term. When you play this program you'll find that Gerry is usually the hardest player to beat.

So what's been created in the poker program is an approximation of these four players. But whereas at a real table on Friday night these flesh-and-blood characters are sometimes quite hard to pin down—sometimes Walter plays a better game than Gerry, sometimes even Marvin has a few surprises up his sleeve—in our program they have been minutely quantified. For the human player sitting at the computer, they represent four different levels of skill, as well as four different players, ranging from Marvin to Gerry (although the human won't know this hierarchy of skill until he's played the game for a while).

Once again, just to get it set in your mind, let's redefine the players:

P$(1) = Marvin (or P(1))
P$(2) = Gerry (P(2))
P$(3) = Walter (P(3))
P$(4) = Betsy P(4))

From this point on I'll be using their names and numbers interchangeably. This anthropomorphization of the program was helpful to me when I wrote it, and now I simply can't shake it. As I said, you might find that a similar anthropomorphization might help you as well in your own strategy games.

## THE MEGALOOP

At this point in the running of the program the human player has anted and watched the cards being dealt. He sees before him his own hand, plus the backs of the cards of the other four players. The player will now study his hand and decide what to bet; likewise, the computer will study *its* hands and decide what to bet. The difference between the human and the computer processes for performing these identical chores is the stuff at the core of understanding how human intelligence differs from computer processing. The way a human thinks allows him or her to look at the hand and analyze it almost instantly; the human is capable of recognizing the various abstract patterns that comprise a hand of poker and classifying them immediately. The human mind does not have to run through all the possibilities from straight flush on down to know what he has; he simply looks at it and recognizes it—and no one really knows how. If we did, we could replicate the process with computers, but since we don't, we're forced to do it another, more tedious way: We must take every possibility and eliminate each and every one, or as they say in the trade, we have to do a lot of number crunching (or in this case, card crunching). Fortunately, the computer is pretty adept at number crunching and does it fairly swiftly, but in no way is this number crunching as swift as the human process of abstract-pattern recognition. It just happens to be swifter than human number crunching, which is why they invented computers in the first place, to perform tedious numerical chores.

The problem we face in this particular crunch is that, despite the speed of the computer, because there's so much crunching to be done this process is noticeably slow, as would have been the shuffling routine if we hadn't sneaked around it. But in this case we don't have a convenient distraction to misdirect the eye of the human player; this time we can't be magicians distracting with our right hands while our left hands are fishing around in our pockets for rabbits. So all the processing contained from line 90 to line 1970, which is quite a lot, will transpire with nothing else going on except the human staring at his hand; our human player will notice a definite pause of 20 seconds or so while he waits for something to happen. Compounding this is the fact that 90 through 1970 is actually a FOR . . . NEXT loop of humongous dimensions: Line 90 reads FOR X = 1 TO 5, which means that we must go through these lines five times, once for each player to figure out what all the hands are. This unfortunate, undisguisable delay is encountered sooner or later in all strategy games of any complexity, and it is just something you have to get used to (with some chess programs at high play levels there's a 24-hour processing delay between moves!). The best you can do is streamline the processing as much as possible to make it as swift as you can.

This programming from 90 to 1970 does a lot of things, but the gist of it is the doping out of what each of the five players is holding. After this section is completed, there will be a phase of betting by all the players. At

one point when the program was finished I did try moving this around a bit: Instead of figuring all the hands at once for a 20-second delay, I tried figuring for one player, then having that player bet, doing it for the next player, then having him bet, and so forth, to see whether five shorter delays were more palatable than one long one. Ultimately, I decided they weren't. This is probably a question of personal taste, so if this kind of option arises in one of your games, I suggest you try it both ways.

The major concern I had in designing this section was the logic of the flow of figuring the hands. I could, of course, have simply laid out a program where the various cards in each player's hand were tried against each and every possible combination of cards, and going with the highest combination, but this would have meant trying some possibilities that could easily be eliminated. For instance, if a player does not hold one pair, there is no way in creation he can hold two pairs. So if we've eliminated one pair we can also eliminate two pairs. Furthermore, if we've eliminated the possibility that any two of his cards match, we can also eliminate three of a kind, four of a kind, and a full house. This kind of logic is simple, of course, but we take it to its fullest extreme, as you'll be seeing shortly. Our chief concern must be not wasting time on the impossible, and these lines of the program come as close to it as I could manage. As I look back on it I see here and there where another arrangement could have sufficed just as well, but the main idea would have remained the same—the least amount of programming to do the most amount of work the fastest. In strategy games where crunching time is of the essence, this must be one of our strongest guiding principles. Let's look at a piece of this part of the program now, taking it one step at a time to keep it manageable.

*FIVE-CARD DRAW*

```
90     FOR X = 1 TO 5
100    IF P(X) < > 0 THEN 1550
130    FOR Y = 1 TO 5
140    V(X,Y) = C(Y + ((X - 1) * 5))
     : V$(X,Y) = C$(Y + ((X - 1) * 5))
     : ST(X,Y) = S(Y + ((X - 1) * 5))
     : ST$(X,Y) = S$(Y + ((X - 1) * 5))
170    NEXT
200    REM  ONE PAIR
210    FOR N = 1 TO 4
220    FOR I = 2 TO 5
240    IF N < > I AND V(X,N) = V(X,I)
         THEN H$(X) = "P" + STR$ (N) +
         STR$ (I)
     : N = 4
     : I = 5
250    NEXT I,N
260    IF LEFT$ (H$(X),1) < > "P" THEN 610
270    IF V(X, VAL ( MID$ (H$(X),2,1))) > =
         11 THEN H$(X) = H$(X) + "J"
280    IF MID$ (H$(X),2,1) = "3" THEN 390
290    IF MID$ (H$(X),2,1) = "4" AND
         RIGHT$ (H$(X),1) < > "J" THEN 610
300    IF MID$ (H$(X),2,1) = "4" THEN 1360
```

Line 90 will come as no surprise, since we already mentioned it. You may be wondering, however, why we bother to analyze five hands, since the computer is playing only four of them. The reason is that the computer will want to know what the human player is holding, not to adjust its own play, but to know whether the human has an opening hand of jacks or better. This will prevent the human from cheating later on. Line 100, which takes us to 1550, is our eliminator. At this point in the game any number of players (including the human) could be out of the game; we want to make sure we don't waste time figuring them too. Line 1550 is the NEXT statement complementing the FOR in line 90—a real megaloop no matter how you slice it.

The loop 130–170 does something very important, which we mentioned earlier in our expanded discussion of arrays. Throughout the program, we want to use the same variables to denote the cards for each player. Up to this point we know that player 1 has the first five cards dealt. He has C(1 through 5), C$(1 through 5), S(1 through 5), S$(1 through 5), and CD$(1 through 5). Player 2 has 6 through 10, player 3 has 11 through 15, player 4 has 16 through 20, and player 5, the human, has 21 through 25. The thing is, in figuring the hands we can't make heads or tails of these C's, S's, and so forth because we want to use our variables horizontally and not vertically. That is, we're concerned with each player's cards from 1 to 5 rather than all the players' cards from 1 to 25. So we create new variables to help us with this. V will be the same as C, V$ will be the same as C$, S and S$ will be ST and ST$ (we don't have to worry about CD$, which we analyze vertically). The formulas in line 140 turn C(1) into V(1,1), C(2) into V(1,2), C(3) into V(1,3), and so on. C(6), player 2's first card, will become V(2,1). By the same token, skipping down a bit, C(19) will become V(4,4). We'll always know which card we're talking about in which player's hand, and it won't matter which player it happens to be. We'll be using the X counter throughout the first part of the array—we'll never throw in a constant—but we'll be maneuvering the Y values from 1 to 5. The formulas on the other side of the equals signs in 140 do this translating for us; they are complicated and, sad to say, mathematically self-explanatory. There's not much of this business in this particular program, but you can encounter this sort of stuff (and probably even more of it and more complicated) in any strategy game. It's basic algebra; just try to remember what they taught you in ninth grade. Finally, the reason we didn't just create V and ST in the beginning, skipping the C and S family entirely, was for more speed in the shuffling routine, which this would have slowed down even further.

The best way to get a feel for these new variables is to look at them in action. The first thing we do in our elimination of possibilities for the hands is see if our player has a pair. If he does, there are numerous possibilities; if he doesn't, there are numerous other possibilities. This determination will send us on the right track right away. In 210–220 we set two nested loops to measure the V, or numeric value, of each of the cards in the hand and look for a match. We want to measure card 1—V(X,1)—against cards 2 (V(X,2)) through 5 (V(X,5)). So we'll start with N = 1 in the outer loop against 2 through 5 in the inner loop. Then we'll measure card 2 against cards 3 through 5 (we'll have already measured it against card 1, hence the truncated looping in 210–220 of 1 to 4 and 2 to 5), card 3 against cards 4 and 5, and

card 4 against card 5. Line 240 does the work for us, making sure that N <> I, then seeing if V(X,N) = V(X,I). If it does, then we do have a pair, so we now can give this hand an H$(X) value. H$(X) will be the name of a hand. At this point in the game we give artificial names that only we and the program will understand; in the case of a pair we will call it "Pxx". The "P" tells us we have a pair, and the "xx"—in this case the string values of N and I—tells us which cards of the five are the pair. Let's say that the hand is this:

JACK CLUB
QUEEN SPADE
JACK DIAMOND
2 HEART
5 SPADE

In this case we would now have an H$(X) of "P13". Knowing what H$(X) is in this fashion allows us to perform a multitude of further calculations based on this information, as you'll see in a second. The remainder of line 240 simply augments the values of N and I so that the loop will be discontinued when we get to 250—we don't want to keep looping after we've found out what we want to know. And the format of line 250 shows how we can combine the NEXTing of two nested loops in one line, inner loop first, then outer loop.

So now we have either an H$(X) of "Pxx" or of "", an empty string. Line 260 sends us to line 610 if we don't have a pair; 610 will begin figuring whether this hand contains a straight or a flush, or parts thereof. Otherwise we go to 270. Here we're going to determine whether we have a high pair or a low pair. At this point we have H$(X) = "Pxx". The left side of the IF equation in 270 pulls out the value of the "x" in the middle—aka VAL (MID$(H$(X),2,1))—and tells us whether V(X,x) is greater than or equal to 11, i.e., jack through ace. If so, we amend H$(X) to "PxxJ" by adding "J" to H$(X). Now we know if we have an opener, jacks or better.

Line 280 is another exclusion like 260, and this time quite subtle. We know that we have five cards and that when we analyzed them we measured the pairs starting with card 1 against card 2 and so forth. This means that if the first card in our pair was card 3, we know that neither card 1 nor card 2 is equal to one another, nor to any other card of the five. In other words, we have either "P34" or "P35" and no chance of two pairs. So we won't bother to figure two pairs, and we'll go straight to 390 to figure three of a kind. Again, this saves us wasting time figuring an impossible combination. Line 290 does more of this. If our hand is "P45" we know we have a low pair and no other matching possibilities, so we scoot out to 610, which, again, is where we figure straights and flushes (we will be throwing away small pairs in favor of four flushes and four straights; i.e., in the hand:

9 DIAMOND
2 CLUB
9 CLUB
4 CLUB
ACE CLUB

we will discard the 9 of diamonds in hopes of the flush improvement). Note that if we have "P45J" we do not go to 610—we won't be throwing away an opener. If we do have this hand, line 300 takes care of it for us (the

logical exclusion from 290) by sending us past the hand figuring into other parts of this segment of the program, about which more later.

What we've just talked about here is very complicated. We've met a lot of new variables, and we've really begun to swim in the deep water of how the poker program works. The rest of this section is more of the same, but it should become clearer as we progress.

## MORE MATCHES

Now we know that we have at least a pair and that there might even be more than a pair. The next step is to figure whether we have two pairs. Although it would be easier figuring three of a kind at this point, in the long run it makes more sense the other way around. You'll see why in a minute.

***FIVE-CARD DRAW***

```
310    FOR N = 1 TO 4
320    FOR I = N + 1 TO 5
330    IF N = VAL ( MID$ (H$(X),2,1)) OR N =
         VAL ( MID$ (H$(X),3,1)) THEN I = 5
     : GOTO 350
340    IF V(X,N) = V(X,I) THEN H$(X) = "2" +
         LEFT$ (H$(X),3) + STR$ (N) +
         STR$ (I)
     : I = 5
     : N = 4
350    NEXT I,N
360    IF V(X, VAL ( MID$ (H$(X),3,1))) = V(
         X, VAL ( MID$ (H$(X),5,1))) THEN H$
         (X) = "FOUR " + V$(X, VAL ( MID$ (H
         $(X),3,1))) + "S"
     : GOTO 1360
370    IF LEFT$ (H$(X),1) = "2" AND MID$ (H$
         (X),3,1) = "1" THEN 530
380    IF LEFT$ (H$(X),1) = "2" THEN
         GOTO 1360
```

Once again we'll have only reached line 310 provided we have an H$(X) of "Pxx" or "PxxJ". Additionally, some impossible situations have already been weeded out in lines 280–300. We can work with this knowledge already in mind. It's a given in the situation that our earlier pair figuring would have given us the first pair encountered starting with the first matched card of the five. So if we held the hand:
10 CLUB
2 HEART
2 DIAMOND
3 SPADE
10 SPADE
our H$(X) would be "P15". So now you know where we stand at this moment.

Lines 310–320 take the nested loop one step farther (or better yet, one

step less) than 210–220. In the earlier loops we eliminated some of the possibilities simply by going from 1 to 4 and from 2 to 5, respectively. Here we discard further impossibilities by defining I as ranging from N+1 to 5, rather than 2 to 5 (actually, we could have done this the first time as well, but a little bit of new information goes a long way). Line 330, of course, merely eliminates duplications we don't want. And 340 goes to the heart of the matter. If this line finds another pair, then our H$(X) changes to "2"+"Pxx"+"xx", or "2Pxxxx". Using the LEFT$(3) of the original H$(X) eliminates the "J" at the end, if there was one, as this information is no longer important. And we now know that the first two little X's represent the numbers of the cards from 1 to 5 that are the first pair, and the second two little X's represent the numbers of the second pair. We will be using those numbers again as time develops.

Line 360 shows why we put the two-pair figuring ahead of the three of a kind: That allowed us this shortcut here. Simply enough, if the V values of both sets of pairs are equal, that means we have two pairs of the same card (two pairs of jacks, for instance): In other words, we have four of a kind. The H$(X) in this line is not an abbreviation for us but a complete spelling out that would look like, for example:

FOUR JACKS

Whenever a hand is finished—i.e., a pat hand where there will be no further changes—we can spell out the name of the hand in full detail, since this name will eventually be printed on the screen, and we want to save ourselves the trouble of renaming it later. As you can see, the end result of this line is a fast ticket out of the hand-figuring to the next part of this routine at 1360.

Lines 370 and 380 give us two more eliminations. The first, in 370, sets up the possibility that we might be holding a full house. The only way this is possible is if H$(X) looks like "2P1xxx". If the first number after the P were a 2 instead of a 1 we'd know that card 1 doesn't match anything. So here we direct programming to the full-house arena if the criterion is met. If it is not and we do not have two pairs, then we simply bow out of hand-figuring in 380 with a branch to 1360. The next line in the program is 390, and the only way we'll get there is if we're holding a pair, either "Pxx" or "PxxJ".

***FIVE-CARD DRAW***

```
390   FOR N = 1 TO 5
400   IF N = VAL ( MID$ (H$(X),2,1)) OR N
        = VAL ( MID$ (H$(X),3,1)) THEN 420

410   IF V(X,N) = V(X, VAL ( MID$ (H$(X),2
        ,1))) THEN H$(X) = "T" + MID$ (H$(
        X),2,2) + STR$ (N)
    : N = 5
420   NEXT N
430   IF LEFT$ (H$(X),1) = "T" OR RIGHT$ (
        H$(X),1) = "J" THEN 1360
450   GOTO 610
```

Here we're looking for three of a kind. We simply set up a loop and look

for a V match to the pair we have already. If we get it, our H$(X) becomes "Txxx", consisting of a "T" plus our first two little X's from the old H$(X) plus a new little X for the third card. Note that in 410, as in all our FOR . . . NEXTs, as soon as we get our condition fulfilled we jump out of the loop. In some cases this is not a logical necessity: In the present case we either have three of a kind or we don't, and all the additional figuring in the world won't change it (if we hadn't already eliminated four of a kind this would not be true). But we do it nonetheless almost by rote. If we've got what we're looking for there's no point wasting time looking for it any further.

Line 430 is the next set of exclusions. If we've got three of a kind or jacks or better, we bounce out of the figuring routine. And if we have a pair lower than jacks, the other possibility, it's off to 610 from line 450 to see if we have a four straight or four flush.

*FIVE-CARD DRAW*

```
530    FOR N = 1 TO 5
540    FOR I = 3 TO 6
550    IF N = VAL ( MID$ (H$(X),I,1))
         THEN I = 6
     : NEXT
     : GOTO 590
560    NEXT
570    IF V(X,N) = V(X, VAL ( MID$ (H$(X),3,
         1))) THEN H$(X) = "FULL HOUSE "
         + V$(X, VAL ( MID$ (H$(X),3,1))) +
         "S"
     : N = 5
580    IF V(X,N) = V(X, VAL ( RIGHT$ (H$(X),
         1))) THEN H$(X) = "FULL HOUSE "
         + V$(X, VAL ( RIGHT$ (H$(X),1))) +
         "S"
     : N = 5
590    NEXT
600    GOTO 1360
```

No, we're not skipping anything here between 450 and 530. Near the end of my debugging I discovered that what once occupied this territory was a routine for figuring a full house if the H$(X) equaled "Txxx". After six months of thinking about it I finally realized that to have reached the three-of-a-kind figuring in the first place we had already eliminated any chance of having a full house. It didn't hurt having these lines here, but it certainly didn't accomplish anything. And it took me a long time even to realize it.

The 530–590 loop will be reached only if we have "2Pxxxx". The inner loop 540–560 eliminates N values equal to the little X's. Lines 570 and 580 give us our full house, either matching the first pair or matching the second pair. Again you'll note the complete spelling out of the hand, e.g.:
FULL HOUSE 7S
And 1360 takes us once again out of the realm of hand-figuring, either with two pairs or a full house.

## THE SNEAKY HANDS

By now we've eliminated all the matches, which was the easy part because either we had them or we didn't. From here on we get a lot more complicated for two reasons: First, the hands themselves are more complicated to analyze, and second, we have to analyze them as pieces as well as wholes. At this point of the game, four cards to a flush or an open-ended straight are quite a good hand, and we have to know if we've got them, just as we've got to know if we have the whole megillah.

To put it mildly, the figuring of a straight is one of the most convoluted operations I've ever programmed. There are a number of ways to do it, and I've used at least three in my time that I can recall offhand. The one here is no better or worse than any of them, and I apologize in advance for the confusion we're going to be going through.

*FIVE-CARD DRAW*

```
610   FOR N = 1 TO 5
620   FOR I = 1 TO 5
630   IF DOUBLECHECK$(X) = "ON" AND N = S4 THEN
        I = 5
    : GOTO 650
635   IF DOUBLECHECK$(X) = "ON2" AND N = S1
        THEN I = 5
    : GOTO 650
640   IF V(X,N) = V(X,I) + 3 THEN S1 = I
    : S4 = N
    : N = 5
    : I = 5
    : NEXT I,N
    : GOTO 760
650   NEXT I,N
660   IF IN(X) = 0 THEN 720
670   IN(X) = 0
680   FOR N = 1 TO 5
690   IF V(X,N) = 1 THEN V(X,N) = 14
700   NEXT
705   IF DOUBLECHECK$(X) = "ON" THEN DOUBLE
        CHECK$(X) = "ON2"
    : GOTO 610
710   DOUBLECHECK$(X) = ""
    : GOTO 910
720   FOR N = 1 TO 5
730   IF V(X,N) = 14 THEN V(X,N) = 1
740   NEXT
750   IN(X) = 1
    : GOTO 610
760   FOR N = 1 TO 5
770   IF V(X,N) = V(X,S1) + 2 THEN S3 = N
    : N = 5
    : NEXT N
    : GOTO 810
780   NEXT
790   IF DOUBLECHECK$(X) = "" THEN DOUBLECH
        ECK$(X) = "ON"
    : GOTO 610
```

```
800    GOTO 680
810    FOR N = 1 TO 5
820    IF V(X,N) = V(X,S1) + 1 THEN S2 = N
     : N = 5
     : NEXT
     : GOTO 850
830    NEXT
840    GOTO 680
850    H$(X) = "4S" + STR$ (S1) + STR$ (S2) +
         STR$ (S3) + STR$ (S4)
860    FOR N = 1 TO 5
870    IF V(X,N) = V(X,S1) + 4 THEN H$(X) =
         "STRAIGHT TO " + V$(X,N)
     : CH(X) = V(X,N)
880    IF V(X,N) = V(X,S1) - 1 THEN H$(X) =
         "STRAIGHT TO " + V$(X,S4)
     : CH(X) = V(X,S4)
890    NEXT
900    GOTO 1140
```

We'll have arrived at 610 only if we have an H$(X) of "" or a low pair. In our straight figuring we're going to start by looking for a situation where the value of one card is separated from the value of another card by a difference of 3. Since that difference could be between any two of the cards, we tediously have to figure them all in our loops at 610–620. Why do it this way? Simple. What we're looking for is a hand where two cards in the middle might give us a four straight, a 3 and a 6, say; or a 9 and a queen. If no two cards of the five are separated by an interval of 3, there is no possibility of either a full straight or a four straight, and we can get right out of this part of the figuring.

We have two other problems, however—one obvious, the other incredibly subtle. The first problem is that, in poker, aces can be either high or low, so we have to figure it both ways. At present, our aces all have a value of 14, so we'll have to change their value to 1 if they don't work out at the 14 level. That's the easy one. The second one, which will happen one time in a million, is a hand where the cards are placed precisely like this:

8 CLUB
JACK DIAMOND
6 HEART
7 CLUB
5 SPADE

Look at that one for a minute, and then look at the programming in this section. Right off the bat you can see that we do have a four straight, the 5 through the 8. No problem. But the way this programming is designed, the first thing the computer will see is the difference between the 8 of clubs and the jack of diamonds. Yes, indeed, this fits the criterion of 640, but when we go to 760 as directed, we will find no cards between the 8 and the jack, which means that there is no straight between them. Alas, unless we plan ahead, we'll miss the straight simply because of the way the cards are arranged. Needless to say, we need to do something about that.

Now, I don't want you to think I thought of this when I first wrote the program. It wasn't until I had run through a couple of hundred debugging hands that I finally saw this problem. We'll talk more about debugging this sort of program in the next chapter and how to find this sort of almost-invisible bug. For the nonce, we do know this problem exists, and line 790 is going to take care of it for us.

In the normal course of events, if the first criterion of 640 is met, the differential of 3, we'll go down to 760. Line 760 will look for another card between the two already set, now known as S4 (the higher one) and S1 (the lower one). If it finds a fit, then on to 810 to look for another piece of the straight. If not, line 790 sets a flag for us and then sends us back to the beginning of this section of programming. Line 630 is looking for that flag, and if it finds it, acts accordingly, discarding half the problem possibility that has arisen. Processing will continue, and line 705 will reset DOUBLE-CHECK$(X) once again to find the other half of the problem (the fit could be off in either direction, S4+3 or S1-3). This is time-consuming processing, but it will arise only occasionally, and it must be done to ensure correct evaluation of the hands.

Meanwhile, there's the (easier?) problem of aces being high or low. We take care of that with the IN(X) flag. (IN as in INversion.) We insert that fix starting at 660, branching to 720 and resetting the V values of any aces, setting the IN(X) flag to 1, and then going back and starting over. If no straight is found, the Vs are reset, as in IN(X), starting at 670.

Where does all this pasta flow, anyhow? Good question. It flows in and out and back and forth and up and down and over and around. It works, and it wasn't easy to get it all straightened out. If you want to follow the logic in action, just program these lines alone with some dummy values (DIM the arrays first), then type TRACE and then RUN.

Anyhow, when you get to the end of all this (line 850), you will have at least a four straight; your small pair, if any, will have been discarded. Your H$(X) will be "4Sxxxx": those little X's will tell you in order which is the lowest card up through the highest. The loop 860–890 will tell you if you have the whole straight, by determining if there's a fifth card one value lower than your first little X, or one value higher than your last little X. CH(X) in these lines is rerunning of the CHeck(X) variable used in our shuffling. For example, two players have straights. The value of CH(X) will tell us which has the higher, and therefore winning, straight. Eventually we will delineate CHECK(X)s for all the hands, and you'll see how we evaluate them at the end of the program.

The only way we'll have gotten this far in the program without bouncing somewhere else is if we have at least the four straight. With either a four or five straight in hand, then, we'll hit line 1140, which will figure whether we have a flush. This is a necessary step regardless of whether we have part or all of the straight. A four flush is a better hand than a four straight, and we would opt for the former if the choice is offered. And there is a very remote possibility that we might have a straight flush. This will be the time to find that out.

Having absorbed all of that, we will see that there isn't much new in the other half of the coin, the dreaded inside straight.

```
910   REM  INSIDE STRAIGHT
920   IF X = 2 OR X = 3 OR X = 5 OR H$(X)
        < > "" THEN 1140
930   FOR N = 1 TO 5
940   FOR I = 1 TO 5
950   IF DOUBLECHECK$(X) = "ON" AND N = S5
        (X) THEN I = 5
    : GOTO 970
955   IF DOUBLECHECK$(X) = "ON2" AND N = S
        1 THEN I = 5
    : GOTO 970
960   IF V(X,N) = V(X,I) + 4 THEN S1 = I
    : S5(X) = N
    : N = 5
    : I = 5
    : NEXT I,N
    : GOTO 1070
970   NEXT I,N
980   IF IN(X) = 0 THEN 990
985   IN(X) = 0
    : GOTO 1030
990   FOR N = 1 TO 5
1000  IF V(X,N) = 14 THEN V(X,N) = 1
1010  NEXT
1020  IN(X) = 1
    : GOTO 930
1030  FOR N = 1 TO 5
1040  IF V(X,N) = 1 THEN V(X,N) = 14
1050  NEXT
1055  IF DO$(X) = "ON" THEN DO$(X) = "ON2"

    : GOTO 930
1060  IN(X) = 0
    : DO$(X) = ""
    : GOTO 1140
1070  FOR N = 1 TO 5
1080  FOR I = 1 TO 5
1090  IF V(X,N) = V(X,S1) + 3 AND V(X,I)
        = V(X,S1) + 2 THEN H$(X) = H$(X)
        + "IS" + STR$ (S1) + STR$ (I) +
        STR$ (N) + STR$ (S5(X)) + STR$ (V(
        X,S1) + 1)
1100  IF V(X,N) = V(X,S1) + 3 AND V(X,I)
        = V(X,S1) + 1 THEN H$(X) = H$(X)
        + "IS" + STR$ (S1) + STR$ (I) +
        STR$ (N) + STR$ (S5(X)) + STR$ (V(
        X,S1) + 2)
1110  IF V(X,N) = V(X,S1) + 2 AND V(X,I)
        = V(X,S1) + 1 THEN H$(X) = H$(X)
        + "IS" + STR$ (S1) + STR$ (I) +
        STR$ (N) + STR$ (S5(X)) + STR$ (V(
        X,S1) + 3)
1120  NEXT I,N
```

```
1130  IF LEFT$ (H$(X),1) < > "I" AND DOUBL
        ECHECK$(X) = "" THEN DOUBLECHECK$(
        X) = "ON"
    : GOTO 930
```

Right off the bat here you can see in line 920 that we'll be figuring this possibility only for players 1 and 4, Marvin and Betsy. Since no one else would dream of drawing to an inside straight, we needn't figure them for it. Further, since we could have arrived at this point for Marvin and Betsy with them actually holding a small pair, we also bump them with that holding by throwing in the H$(X)<>"" bit. I want you to look at this line for a minute, because there's a devil here, and it reminds me of an exam I once took in high school biology. On that exam, which was multiple-choice, there was a question something like this:

29. WHICH OF THE FOLLOWING IS FALSE?
    A. an amoeba is not free-moving
    B. a protozoa can be multicellular
    C. an amoeba is either animal or vegetable
    D. all of the above
    E. none of the above

I assure you that the groans shook the whole building when this question popped up. It is a model of disjointed logic, consisting not only of NOTs but also of NOT NOTs, NOT NOT NOTs, and even a NOT NOT OR NOT NOT. Knowing the correct answers was no clue, because if they were all correct, then the correct answer would have been E, none of the above. This is the kind of nonsense that drives students into the streets.

Anyhow, line 920 is a mild example of this disjointed logic. In Boolean terms it looks like this: IF X OR Y OR Z OR NOT A THEN B. Not so confusing, perhaps, but where the first three criteria are straightforward, the fourth is a NOT mixed in for good measure. The logic is simple enough here to work and make sense, but this kind of programming line can turn you old before your time if the conditions are complex enough. Beware of such logic, however accurate, in your programs. Sometimes it's easier to write a couple of extra lines of program than to try to figure out these monsters.

This digression notwithstanding, the inside-straight figuring mostly just recapitulates the regular straight figuring, this time looking for a gap of 4 with a missing card in between, e.g., 4 and 8, with the other cards a 5, 6, and queen—or whatever. All the IN(X) and DOUBLECHECK$(X) business is in place as before. If we do find an inside straight, we will give it an H$(X) like this: "ISxxxxx". The first four little X's tell us which are the cards we're keeping, in order. The fifth X tells us the *value* of the card we'll need to draw to fill in the straight. You'll note in 955 the old precept about Applesoft Basic concerning itself only with the first two characters of a variable name. Even though the whole name is registered if we've typed it in, only the first two letters count, so in these lines DOUBLECHECK$(X) and DO$(X) are in fact interchangeable. It's not necessarily a good practice to use two names for the same variable like this, but DOUBLECHECK$(X) was so long I just couldn't resist abbreviating it.

At the end of this segment we will slide right into the flush-figuring

section, since both Marvin and Betsy would much rather have a four flush then a dreaded inside straight.

*FIVE-CARD DRAW*

```
1140  FOR N = 2 TO 5
1150  IF ST(X,N) < > ST(X,1) THEN N = 5
    : NEXT
    : GOTO 1200
1160  NEXT
1170  IF LEFT$ (H$(X),4) < > "STRA" THEN H$
        (X) = "FLUSH " + ST$(X,1) + "S"
1180  IF LEFT$ (H$(X),4) = "STRA" THEN H$(X
        ) = "STRAIGHT FLUSH"
1190  GOTO 1360
1200  FOR N = 1 TO 5
1210  FOR N2 = 1 TO 5
1220  FOR N3 = 1 TO 5
1230  IF N2 = N3 OR N = N3 OR N = N2
        THEN 1250
1240  IF ST(X,N) = ST(X,N2) AND ST(X,N)
        = ST(X,N3) THEN H$(X) = H$(X) + "3F
        " + ST$(X,N)
    : F(X) = N
    : F2 = N2
    : F3 = N3
    : N = 5
    : N2 = 5
    : N3 = 5
    : NEXT N3,N2,N
    : GOTO 1270
1250  NEXT N3,N2,N
1260  GOTO 1360
1270  FOR N4 = 1 TO 5
1280  IF N4 = F2 OR N4 = F3 OR N4 = F(X) THEN
        1300
1290  IF ST(X,N4) = ST(X,F(X)) THEN H$(X) =
        "4F" + ST$(X,F(X))
    : N4 = 5
    : NEXT
    : GOTO 1360
1300  NEXT
1310  IF X < > 1 AND LEFT$ (H$(X),2) = "3F"
        THEN H$(X) = ""
    : GOTO 1360
1320  IF LEFT$ (H$(X),2) = "3F" THEN 1360
1330  FOR I = 1 TO LEN (H$(X))
1340  IF MID$ (H$(X),I,2) = "3F" THEN H$(X)
        = LEFT$ (H$(X),I - 1)
1350  NEXT
```

This is not a complicated figuring, especially compared to the straight. If we can survive the loop at 1140–1160 without branching to 1200, we definitely have a full flush. If we also have a straight, then we have a straight flush. Lines 1170–1180 take care of those possibilities.

More likely, at this point we have less than a full flush. Lines 1200–1250 give us a three-looper, looking for (shiver!) a "3F" H$(X). Marvin, indeed, would play a three flush if he had nothing better. Note how 1250 combines all three levels of the loop in one neat NEXT. The reason we set F(X) will become clear later, so don't worry about it now. Keep in mind that we have to initialize new variables when we're in the middle of a loop like this because the original variables of N, N2, and N3 will be augmented by the loop itself. If we succeed with our three flush we'll go on to a four flush, in 1270–1300. And finally, lines 1310–1350 clean up the hands, making sure we have kept the hands of the highest value. And that's the end of the hand-figuring section of the program.

## NAMING THE HANDS

We've already given our hands H$(X) names, of either of two varieties. The first variety has been the shorthand name ("Pxx" or "ISxxxxx") for our own internal figuring purposes. Some of those purposes we've already discovered; the rest we'll see later. The second variety of names has been the fully spelled out ("FOUR QUEENS", "FLUSH CLUBS") for the external purpose of eventually printing them on the video screen. But we have to know more about the hands than just what they are—we also have to know how strong they are relative to each other. So we establish a hierarchy, rating the hands from 0 to 13.

***FIVE-CARD DRAW***

```
1360  REM   ASSIGN H(X) VALUES
1380  IF H$(X) = "" THEN H(X) = 0
1390  IF LEFT$ (H$(X),2) = "IS" THEN H(X) =
        1
1400  IF LEFT$ (H$(X),2) = "3F" THEN H(X) =
        2
1410  IF LEFT$ (H$(X),2) = "4S" THEN H(X) =
        3
1420  IF LEFT$ (H$(X),2) = "4F" THEN H(X) =
        4
1430  IF LEFT$ (H$(X),1) = "P" THEN H(X) =
        5
    : IF LEN (H$(X)) > 3 THEN H(X) = 6
1450  IF LEFT$ (H$(X),2) = "2P" THEN H(X) =
        7
1460  IF LEFT$ (H$(X),1) = "T" THEN H(X) =
        8
1470  IF LEFT$ (H$(X),10) = "STRAIGHT T" THEN
        H(X) = 9
1480  IF LEFT$ (H$(X),2) = "FL" THEN H(X) =
        10
1490  IF LEFT$ (H$(X),2) = "FU" THEN H(X) =
        11
1500  IF LEFT$ (H$(X),2) = "FO" THEN H(X) =
        12
1510  IF LEFT$ (H$(X),10) = "STRAIGHT F" THEN
        H(X) = 13
```

```
1520  IF REDO = 1 THEN RETURN
1540  ON X GOSUB 1570,1630,1710,1900
1550  NEXT
1560  RETURN
```

These lines do this rating job for us. A hand with nothing gets a 0, an inside straight gets a 1, a three flush a 2, and so forth. This is straightforward drudge programming, but at least it's short. The only cute thing we do is measure the LEN of any H$(X) beginning with "P"—if it's more than 3 (line 1430) we know we have at least a pair of jacks, "PxxJ", so H(X) = 6. Once again in this segment we're using mirrored variables, this time H$(X) and H(X). These two are so inextricably related that it makes utmost sense to bind them like this. Of course, one is a string variable and the other a real-number variable, so as far as the computer is concerned they are worlds apart, but we know better.

Line 1520 introduces a factor we'll let ride for a minute; we'll come back to the variable REDO shortly. Line 1540 branches into separate subroutines depending on the value of X to determine each player's betting strategy for this segment of the game. Note that although X could have up to five values, there are only four GOSUBs listed. In an ON statement, if the value of X matches up with one of the possibilities (and here there are four possibilities), then it will GOSUB (or GOTO) as directed. If there is no correct possibility, in this case X = 5, then the computer will simply ignore the ON . . . GOSUB statement and continue processing on the next line. We will be using this fact again later, and it is important to remember when using ONs to make sure you have enough directions for each variable, because if you don't, the computer will just keep on going (which in this case is exactly what we want). As for these particular GOSUBs, they're worth a whole section to themselves, immediately following this one. Please stay tuned.

Last, we have the NEXT statement at 1550. NEXT what? you might ask. It's NEXT X. (It's faster not to include the variables in a NEXT statement if you can avoid them. If you use them, the computer will make sure it's the right one; if you don't, it will just do the NEXT whatever comes next. The best rule is to use the variable names only when you as programmer feel you need them to keep things straight in your own head.) At this point in the program, you'll go back and do the whole thing over again for the next player. And if all the players have been taken care of, then it's good-bye, *amigo,* and back to the traffic-control section at 10000.

## CREATING THE BETTING ALGORITHMS

At this point we have determined the holdings of each player, but we haven't done anything about them. It makes no difference yet whether Betsy is going to draw to an inside straight, as long as we know whether she's got that inside straight. But in a minute we're going to be betting, and that tactical information of who's going to be playing what, plus that strategic information of how much they're going to be betting on what, are going to become very important indeed. And while I was very cavalier in an earlier chapter

on strategy and tactics and stated that strategy was the way one handled one's money and tactics was the way one handled one's cards, the programming does at times overlap, as we'll see now. But for the most part tactics are much more easily accommodated by the program than the strategy will be. If a given player doesn't have a hand of a certain value, he will simply drop out. 'Nuff said. Strategy is a more complicated matter.

In the real world, where nothing but human beings are sitting down at the poker table, the factors governing betting are multitudinous. We tend not only to bet our own hands but also to bet against the other players. If Marvin, who always stays in no matter what he's holding, makes a small bet, we do not worry about it. On the other hand, if Marvin, who never bluffs, makes a big bet, we tend to believe him. We respond to both who is betting and what he is betting. And in real poker we don't have too much difficulty keeping track of all the different players; the human brain has no problem monitoring four or five shifty characters, including itself. But in generating a game of computer poker we have to have a different focus for our intentions. Almost by definition our computer game is man vs. machine. We are not trying with each of our four computer-controlled players to beat each of the other computer players, which means we will be giving them no inside knowledge on each other. That is, just as the human player won't know that Betsy is a big bluffer, neither will Gerry. The difference is, while the human player will eventually learn this fact, the computer Gerry won't. Essentially each of the four computer players is playing head-on against the human, and the hell with everything else. This is a common trick in computer games. As a rule, it is almost impossible to program a single opponent as good as the average human in any game, shoot-'em-up or strategy. So what game designers do is bring in the strength of numbers against the human's strength of wit. Maybe we can't program one spaceship to outshoot the human's spaceship, but if we use 50 or 60 spaceships against the human's one, then maybe the computer will have a chance. Why do you think there are so many Space Invaders up there, and only one of you? You're just too good for any program unless that program uses brute strength.

In "Five-Card Draw" our brute strength is our four players vs. the human's one. And that brute strength is going to operate ultimately on the basis of three variables: DF(X), BF(X), and RF(X). You can think of them as Drop Factor, Bet Factor, and Raise Factor, and they will work very simply. Each one is expressed in terms of betting dollars: the Drop Factor, or DF(X), is the amount of money bet that will cause our computer-operated player to drop out—i.e., if there's enough money against him, player X will simply fold his hand. The Bet Factor, or BF(X), is the amount of money the player will open on a given hand. And the Raise Factor, the RF(X), is the amount of money the player will raise on a given hand. Let's take each player one by one.

***FIVE-CARD DRAW***

```
1570  REM  MARVIN
1580  IF H(1) < 8 THEN BF(1) = 1
    : RF(1) = 0
```

```
1590  IF H(1) = 8 OR H(1) = 9 THEN BF(1) =
        2
    : RF(1) = 2
1600  IF H(1) > 9 THEN BF(1) = 3
    : RF(1) = 5
1610  DF(1) = 21
1620  RETURN
```

As you know, Marvin is going to stay in no matter what. He is a player of blunt tactics and minimal strategy. If he has a pair or two pairs (an H(1) of 7 or 8—line 1580) he will open for $1, and he won't raise. On three of a kind or a straight he'll up that to $2, with an RF(1) (Marvin is P(1)) of $2. With any pat hand he'll open for 3 and then break open his tell-all poker face by raising the maximum of $5. And he will not drop out, even if he is holding nothing (1610). Since $5 is the maximum bet and there is a three-bet limit, the maximum amount of money that can be on the table (less antes, which is how we'll be figuring it) is $20. Marvin's Drop Factor is $21. Good luck, Marvin.

***FIVE-CARD DRAW***

```
1630  REM  GERRY
1640  IF H(2) < 3 THEN DF(2) = 0
1650  IF H(2) = 3 OR H(2) = 4 THEN RF(2) =
        0
    : DF(2) = 6
1660  IF H(2) > 4 AND H(2) < 8 THEN BF(2) =
        1
    : RF(2) = 0
    : DF(2) = 4
    : IF H(2) > 5 THEN DF(2) = 14
1670  IF H(2) = 8 OR H(2) = 9 THEN BF(2) =
        2
    : RF(2) = 1
    : DF(2) = 21
1680  IF H(2) = 10 THEN BF(2) = 3
    : RF(2) = 3
    : DF(2) = 21
1690  IF H(2) > 10 THEN BF(2) = 3
    : RF(2) = 5
    : DF(2) = 21
1700  RETURN
```

Again, Gerry is our best player. He won't even play the game through the draw if he isn't holding at least a four straight (line 1640). Even with a four straight or four flush he has a conservative DF(2) of 6 (line 1650). It's only when he is holding an opener (end line 1660) that his DF(2) is high enough to keep him in the game under any kind of serious betting. And even when he has a really good holding he doesn't tip his hand too early. He'll raise, all right, but he's still conservative; look at those RF(2)s in 1670 and 1680. He wants to keep his opponents in the game. It's only when he has a minimum hand of a full house that he starts getting nervous and raising the limit. Even Gerry isn't perfect.

```
1710  REM  WALTER
1720  IF H(3) < 3 THEN DF(3) = 0
1730  IF H(3) = 3 OR H(3) = 4 THEN RF(3) =
        3
    : DF(3) = 6
1740  IF H(3) < 8 AND H(3) > 4 THEN BF(3) =
        1
    : RF(3) = 0
    : DF(3) = 4
    : IF H(3) > 5 THEN DF(3) = 10
1750  IF H(3) = 8 THEN BF(3) = 2
    : RF(3) = 2
    : DF(3) = 21
1760  IF H(3) > = 9 THEN BF(3) = 3
    : RF(3) = 3
    : DF(3) = 21
1770  IF H(3) = 6 THEN GOSUB 1800
1775  IF H(3) = 8 THEN GOSUB 1850
1780  RETURN
1800  FOR I = 1 TO 5
1810  IF I = VAL ( MID$ (H$(3),2,1)) OR I =
        VAL ( RIGHT$ (H$(3),1)) THEN 1830
1820  IF V(X, VAL ( MID$ (H$(X),2,1))) < >
        14 AND V(3,I) = 14 THEN H$(3) = H$(
        3) + "K" + STR$ (I)
    : BF(3) = 2
    : RF(3) = 4
    : DF(3) = 21
1830  NEXT
1840  RETURN
1850  FOR I = 1 TO 5
1860  IF I = VAL ( MID$ (H$(3),2,1)) OR I =
        VAL ( MID$ (H$(3),2,2)) OR I =
        VAL ( RIGHT$ (H$(3),1)) THEN 1880
1870  IF V(X, VAL ( MID$ (H$(X),2,1))) < >
        14 AND V(3,I) = 14 THEN H$(3) = H$(
        3) + "K" + STR$ (I)
    : BF(3) = 1
    : RF(3) = 4
    : DF(3) = 21
1880  NEXT
1890  RETURN
```

Because Walter is cagey, his factors are the longest to program. Like Gerry, he'll drop with a stinker hand, but look at the RF(3) in line 1730. If Walter is holding a four straight or four flush, he's going to push up that betting quite a bit—that's what betting on the come is all about. After that, he gets sneaky. Note that he doesn't push those RF(3)s on the big hands. When Walter is sitting pretty, he's going to make sure the other players are sitting in there with him.

The exception we have with Walter is his willingness to keep a kicker; hence the further subroutines at 1770–1775. If Walter is holding an ace with either a high pair or three of a kind, he's going to hold onto it in the hope of fooling his opponents. And his factors will change accordingly. The pro-

gramming in these lines is similar to the earlier match figuring. Tricky guy, that Walter.

*FIVE-CARD DRAW*

```
1900  REM  BETSY
1910  IF H(4) < 3 THEN DF(4) = 0
1920  IF R(4) > 50 AND H(4) < 3 THEN RF(4) =
        0
    : DF(4) = 6
1930  IF H(4) > 2 AND H(4) < 6 THEN RF(4) =
        0
    : DF(4) = 6
1940  IF H(4) = 6 THEN BF(4) = 1
    : RF(4) = 0
    : DF(4) = 9
1950  IF H(4) = 7 OR H(4) = 8 THEN BF(4) =
        2
    : RF(4) = 2
    : DF(4) = 21
1960  IF H(4) > 8 THEN BF(4) = 3
    : RF(4) = 3
    : DF(4) = 21
1970  RETURN
```

And finally there's Betsy, lone female at the table. She's a gut player and an odd one. For her, the size of her bankroll is just as important as the cards in her hand. Look at 1910–1920. In 1910 she is tempted to drop with a nothing hand, but in 1920, if she notices that she has more than $50 in chips (R(4)>50), she will decide to chance it after all, provided the further stakes aren't too high—that DF(4) is still pretty low. Following that, she's relatively conservative . . . but wait until you see her at the second round of betting.

## BETTING THE HANDS—COMPLETING THE BETTING ALGORITHM

Now we know what everybody has in his or her hand, and we've established a pattern for betting those hands. We're ready at this point for the actual betting. And believe me, it's as complicated a segment of the program as any other, if not more so. Not because there's anything exotic about the logic—far from it—but because we have so much to keep track of at the same time. If you decide to rewrite this game without a betting limit or a limit to the number of raises, or anything along those lines, be my guest. I'll send you a get-well card at the mental institution.

*FIVE-CARD DRAW*

```
8000  REM  FIRST ROUND OF BETTING
8010  POKE 49168,0
    : NB = 0
    : RD = 1
    : OP = D + 1
    : IF OP = 6 THEN OP = 1
```

```
8050  IF P(OP) = 0 THEN 8110
8060  OP = OP + 1
    : IF OP = 6 THEN OP = 1
8070  RD = RD + 1
    : IF RD = 6 THEN POP
    : D = D + 1
    : GOTO 8090
8080  GOTO 8050
8090  IF D = 6 THEN D = 1
8095  IF P(D) = 1 THEN D = D + 1
    : GOTO 8090
8100  GOSUB 29170
    : HOME
    : GOTO 10020
```

As we mentioned earlier, when you use that POKE/PEEK pair to obtain input from the keyboard, you have to be careful with subsequent INPUT statements. After POKEing 49168, because there's been no subsequent RETURN, the computer has a funny way of thinking you've INPUT something when you haven't; sometimes you'll see the results of this on the screen like this:

WHAT'S YOUR BET?
WHAT'S YOUR BET?

The computer will print the INPUT statement in your program twice because there're some undesirable data still lingering at those INPUT addresses. The nature of that data is such that the computer's native error-catching routine knows there's something wrong; hence the repeat of the INPUT statement showing that the computer has internally cleared its INPUT deck. This explains the POKE that inaugurates line 8010, to prevent this from happening.

Right at the start we have three new variables to digest here, and be forewarned that we're heading into the variable badlands. The first one is NB, as in Number of Bet. Remember, there's a three-raise limit, so if we start with an NB of 0, the maximum value it can attain is 3 on the third raise. The second variable is RD, as in RounD. A round consists of five bets—i.e., each player making one bet. At the end of any round of five bets without any raises, the betting will end, whereas any legal raise would start a new round and RD would be reset. And finally there's OP, which is going to be our counter variable so we'll know who's betting at a given time. In poker, the betting begins with the player on the dealer's left, so we start with an OP value of OP=D+1. Since we have only five players, and player 1 happens to be the one to the left of player 5, we twist back to 1 from 6 as at the end of this line, if necessary. You'll see variations of this theme repeated endlessly in the betting sections.

The first thing we have to do in this segment is make our eliminations. If a player is out of the game, we don't want him betting. Line 8050 sends us down to the actual betting, provided player OP (P(OP)) is in the game. This may seem contrary to the mob-at-the-turn-in-the-road rule, but the betting segment runs without time problems, so it doesn't matter here. And as the game progresses and players begin to drop out, it may become the mob that is *not* in the game, rather than the other way around. Play it and see.

Following this we start to get detailed. We augment OP in line 8060, and then we augment RD in 8070. If we've gotten to this point and RD is 6, that means that no one opened, so we POP out of the subroutine altogether, and then we augment D, the dealer, to move the deal along. Then we make sure that player D is still in the game (a P(D) = 1 would mean permanently out), and if he is no longer with us, we augment D once again. When we finally get a D still in the game we GOSUB 29170, which will clear all the variables that need clearing, and then we go back to 10020 in the traffic-cop routine, which is a new shuffle and ante. These lines 8060–8100 are what I think of as bread-and-butter programming, and they're much easier to explain than to do. They require you to think of a lot of things at once—the movement of the deal, the movement of the round, the movement from 6 to 1, the elimination of nonplaying players—which are very hard to keep in mind when you're programming the first time through. So be prepared for problems in these allegedly simple areas: They can be absolute killers.

***FIVE-CARD DRAW***

```
8110  ON OP GOSUB 7000,7040,7080,7120,7160
8120  IF OP < > 5 THEN 8250
8130  INPUT "WHAT'S YOUR BET?";B(OP)
8140  GOSUB 7180
    : PRINT G$
8150  IF H(5) < 6 AND B(OP) > 0 THEN
        GOSUB 7160
    : PRINT "YOU CAN'T OPEN!"
    : OP = 1
    : GOTO 8070
8160  IF B(OP) > 5 OR B(OP) > R(5) OR B(OP)
        < 0 OR B(OP) < > INT (B(OP))
        THEN PRINT "ILLEGAL BET"
    : GOSUB 63000
    : GOSUB 7160
    : GOTO 8130
8170  IF B(OP) = 0 THEN OP = 1
    : GOTO 8070
8240  R(5) = R(5) - B(OP)
    : PT = PT + B(OP)
    : LB(OP) = B(OP)
    : GOTO 8330
8250  IF H(OP) > = 6 AND R(OP) > = BF(OP) THEN
        PRINT B$(1) + STR$ (BF(OP)) + G$
    : R(OP) = R(OP) - BF(OP)
    : PT = PT + BF(OP)
    : LB(OP) = BF(OP)
    : GOTO 8330
8260  IF H(OP) > = 6 THEN PRINT B$(1) + "1"
        + G$
    : R(OP) = R(OP) - 1
    : PT = PT + 1
    : LB(OP) = 1
    : GOTO 8330
```

```
8270  PRINT B$(8) + G$
    : GOSUB 63000
    : GOTO 8060
8330  BE = LB(OP)
    : STARTER = OP
    : RD = 1
    : GOSUB 63000
```

The next thing we do is get the first, opening bet. Line 8110 places our text on the appropriate chat lines for each player, and 8120 once again disregards the-mob-at-the-turn-in-the-road rule, but again not with a noticeable increase in processing time. So let's start with the human player first, as the program does.

First we get the input of the human's bet, which we'll call B(OP), B as in Bet. Line 8140 rings the bell for us, and you'll note that it's preceded by another placement of the cursor. Printing G$ is just like printing anything else: If we don't do anything to stop it, it will automatically start flush left at a new line below the last printed line, in this case scrolling up all the material above it, something we definitely don't want with our carefully designed playing screen. Hence the tabbing GOSUB.

Lines 8150–8160 eliminate the incorrect bets. Line 8150 simply checks to make sure that if player 5 has opened, he has the cards to do it, an H(5) of greater than 5 (a 6 would mean jacks or better). If this is not the case, player 5 is admonished with a YOU CAN'T OPEN, OP is augmented (so to speak) from 5 to 1, and we go to 8070 to move things along to the next player. Line 8160 looks for other problems. In order they are: a bet greater than $5, a bet greater than the player's bankroll (R (5)), a bet less than $0, or a bet that is not an integer (e.g., $3.98). In any of these cases the decks are cleared and the player is asked once again for his bet in line 8130.

Having eliminated all the wrong possibilities, we now proceed to the correct ones. If player 5 bets $0, we go on to the next player (8170). Otherwise, his bankroll is decreased by the amount of his bet, the pot is increased by the same amount, and a new variable, LB(OP), is born, made equal to the amount of his bet. LB(OP) is going to mean the Last Bet of a given player . . . for a very simple reason. As the betting progresses and raises are made, when it comes the turn of a given player to bet, he must bet all the raises that have occurred since his last bet *minus* that last bet. Let's say player 1 bets $2. Then player 2 raises him $3. The bet to the rest of the players is $5, but when we get back to player 1, the bet is $5-LB(OP), $5 – $2 or $3. We need this information both internally, for bankroll and pot adjusting, and externally, to keep the human apprised of what's going on. The variable that would represent the *total* bet, in this example the $5, we will call BE.

Line 8250 begins handling the computer players. If the hand is an opener and the player's bankroll is larger than his BF, then he will bet his BF, with all the other variables being raised or lowered accordingly. If he's low on funds, line 8260, he'll bet just $1. Computer players will be allowed to "go light" once a game is in progress—i.e., play without enough money on hand—but if they don't win that hand, it's the end of them (the computer will automatically trace their bankrolls in negative numbers as needed). If

player OP doesn't have an opener, he passes in line 8270, and we move on to the next player. Note that it is possible, just as in real life, for all five players not to have an opening hand. A little dull, perhaps, but the pot does keep growing if that is the case, which helps a little bit in making up for this lack of good cards.

The aforementioned BE is introduced in 8330, again BE being the total bet on the board. If one player bets $2 and the next player raises $3, then BE would be 5. If the next player raises 1, then BE would be 6, and so on. Although in some cases BE will translate directly to a bet as the players might see it, in others it will be keeping track of something the players have long since forgotten. It's the relationship BE−LB(OP) that's the important one. The variable STARTER—we've seen it in our info-box as ST—gives us the number of the player who has opened. It is important to know this because it is the STARTER who will open the second round of betting, not the player to the dealer's left who had been given the first opportunity to open in round 1. And after a pause, the bets continue.

***FIVE-CARD DRAW***

```
8340  REM  BETS CONTINUE AFTER OPENER
8350  OP = OP + 1
    : IF OP = 6 THEN OP = 1
8360  IF P(OP) < > 0 THEN RD = RD + 1
    : GOTO 8530
8370  GOSUB 31000
8380  ON OP GOSUB 7000,7040,7080,7120,7160
8390  IF OP < > 5 THEN 8480
8400  INPUT "WHAT'S YOUR BET?";B(OP)
8410  GOSUB 7180
    : PRINT G$
8420  IF B(OP) > R(5) OR B(OP) > BE - LB(OP
        ) + 5 OR B(OP) < 0 OR (B(OP) > BE -
        LB(OP) AND NB = 3) OR B(OP) < > INT
        (B(OP)) THEN GOSUB 7160
    : PRINT "ILLEGAL BET" + G$
    : GOSUB 63000
    : GOSUB 7160
    : GOTO 8400
8430  IF B(OP) < > 0 AND B(OP) < BE - LB(OP
        ) THEN GOSUB 7160
    : PRINT "ILLEGAL BET" + G$
    : GOSUB 63000
    : GOSUB 7160
    : GOTO 8400
8440  IF B(OP) = 0 THEN P(5) = 10
    : GOSUB 40000
8450  R(5) = R(5) - B(OP)
    : PT = PT + B(OP)
    : RD = RD + 1
8460  IF B(OP) > BE - LB(OP) THEN RF(OP) =
        B(OP) - BE - LB(OP)
    : BE = BE + RF(OP)
    : NB = NB + 1
    : RD = 1
```

```
8470  LB(OP) = BE
    : GOTO 8520
8480  IF DF(OP) < = BE THEN PRINT B$(9)
        + G$
    : RD = RD + 1
    : P(OP) = 10
    : GOSUB 40000
    : GOTO 8520
8490  IF NB < 3 AND RF(OP) > 0 THEN
        PRINT B$(3) + STR$ (RF(OP)) + G$
    : R(OP) = R(OP) - RF(OP) - BE + LB(OP)
    : PT = PT + RF(OP) + BE - LB(OP)
    : RD = 1
8500  IF NB < 3 AND RF(OP) > 0 THEN BE = BE
        + RF(OP)
    : LB(OP) = BE
    : NB = NB + 1
    : GOTO 8520
8510  RD = RD + 1
    : PT = PT + BE - LB(OP)
    : R(OP) = R(OP) - BE + LB(OP)
    : LB(OP) = BE
    : PRINT B$(2) + G$
8520  GOSUB 63000
8530  IF RD < > 5 THEN 8350
```

At the beginning of this section we simply do a little incrementing as before, and then we rerun the 31000 subroutine on the screen. We discussed 31000 in our chapter on formatting, but it might make sense now to look at it again. This is the subroutine for displaying the ongoing play information. If a STARTER (or ST) exists, which it now does, the name of that player is printed. Below that is the name of the dealer, followed by a line "THE BET IS " BE−LB(OP), followed by the number of raises (NB, which will be 0 until a raise is encountered), followed by the amount of the pot and the amount of the human player's bankroll. This is nice information to have at all times, which is why we put it there. And we will rerun 31000 every time there's a possible change in any of these variables.

The format of this part of the betting section is similar to the actual opening but a bit more involved to cover some new possibilities. There are more possible illegal bets for the human, for instance; they take up two lines (8420–8430), and they are, in order: bet greater than bankroll, raise greater than $5, bet less than $0, a raise when there have already been three raises, bet not an integer, and (in 8430) low a bet other than $0 (which would mean a dropping out by the human player)—that is, a bet less than the correct amount he must bet. If his bet does equal $0 (line 8440), he gets a P(5) value of 10, meaning he has dropped out of this hand, and his cards are removed from the screen. And even though he is the only real player involved, the game will continue without him—a good time either to go to the bathroom or to take notes on how the other players conduct their play.

As before, if player 5 bets, the appropriate moves are made, this time in 8450. But now there's an extra wrinkle: What if player 5 decided to raise? In that case 8450 would be neither complete nor entirely accurate, so we have to handle that in 8460. If the bet is greater than BE−LB(OP), we

introduce an RF for player 5, the difference between what he did bet and what he was supposed to bet. That difference is the amount of his raise. Then the BE is raised accordingly, NB is incremented, and the RD is reset at 1. But whether he raised or not, his new LB(OP) will be equal to the BE, whatever it is; hence 8470.

Now we go back to our nonhuman players. The first thing we handle is when they're going to drop out, when their DF is less than the BE in 8480. Lines 8490–8500 cover the situations where our computer player is going to raise. Provided that the three-raise limit hasn't been reached and that the player's RF is greater than 0, then these lines handle his roll, the pot, the round, the BE, the LB(OP), and the NB, respectively (in two separate lines, for programming sanity). If he's going to stay in but not raise, then 8510 does the job. Line 8530 figures whether there are any players left. If so, back to the beginning. If not . . .

***FIVE-CARD DRAW***

```
8540  BE = 0
    : LB(OP) = 0
    : GOSUB 31000
8550  SOFAR = 0
8560  FOR N = 1 TO 5
8570  IF P(N) < > 0 THEN SOFAR = SOFAR + 1
8580  NEXT
8590  IF SOFAR < > 4 THEN SO = 0
    : RETURN
8600  FOR N = 1 TO 5
8610  IF P(N) = 0 THEN WINNER = N
8620  NEXT
8630  GOSUB 28250
    : POP
    : GOTO 10140
```

It would seem that we're ready now to go back to the traffic cop and proceed to the draw part of the game, but there's still one thing left to decide. It is not impossible that all but one player has dropped out of the game (it is impossible for all five of them to have dropped out—the DFs were designed to prevent that). So at this point we might already have a winner by default. This part of the program checks for that possibility.

We're going to use the variable SOFAR (or SO) to determine this, and we'll simply loop through all the players in line 8570, incrementing SOFAR/SO every time a player is out. Then, if SOFAR equals 4, we know we have one player left. That player's number on P(N) is made the WINNER (a variable), and processing GOSUBs to the winner's circle scenario, then POPs out of this one, and a new hand is shuffled and anted back at 10140. We'll cover the winning scenario later in this chapter.

## GETTING THE DRAW CARDS

Now that the first round of betting has transpired, the players are ready to take their draw cards. The maximum draw any player can take is three cards;

although some variations of draw allow a player to take up to five cards, we are going to be much more strict in our game.

At this point we know two things about each hand, its H$(X) and its H(X). In this part of the program you'll see why most of the H$(X)s were given the names they were and how exactly those names are used within the program (a few of the names were designed to aid in debugging, but more about that later). This is a very long part of the program, so settle in for a while.

*FIVE-CARD DRAW*

```
2000  REM  GET DRAW CARDS AND FIGURE HANDS
2010  DR = 26
    : X = D + 1
    : IF X = 6 THEN X = 1
2030  IF P(X) < > 0 THEN 4190
2035  FOR I = 0 TO 3
    : NC(I) = 0
    : NEXT
2040  ON X GOSUB 7000,7040,7080,7120,7160
2050  IF X = 5 THEN 3470
2060  ON H(X) GOTO 2320,2520,2910,3000,3070
        ,3140,3220,3310
2070  IF H(X) > 8 THEN 3460
```

The draw section of the program goes from 2000 to 4230, but most of this breaks down to small segments, depending on the H(X) of the particular hand. The first thing we do is initialize a new variable, DR. DR (as in DRaw card) starts at 26 and is going to act as our counter variable for tracking down the cards dealt to each player. You'll remember that back in the shuffle section we created 40 cards at random; so far we have used 25 of these, five to each player, from CD$(1) to CD$(25) (and all the other C's and S's and so forth that go with each card). All we want to do here is take the next card off the pack starting with number 26; DR helps us do that successfully.

As in real poker, the first player to take his draw cards is the first player to the dealer's left still in the game. Hence 2010 sets our ubiquitous X at D + 1, and as usual X is demoted to 1 if it reaches the "wrong" value of 6. Line 2030 simply sends us to a section of the program where X is augmented if P(X) is not playing in this hand. Line 2035 clears all the values of a variable we'll be looking at very shortly. Line 2040 is where we set our cursor at the chat lines, and we'll discuss how we handle the human player (in line 2050) after we dispose of the nonhuman ones. Here the mob at the turn in the road was handled correctly: Of five possible x values, four pass through, and only one has to branch. The processing speed of the whole draw section varies a lot, depending on exactly what the H$(X) was to begin with, anywhere from practically instantaneous to a few seconds. Nothing to get upset about, but we don't want to make it any worse, hence our putting this segment of the program down low at 2000, and a close eye on the structure as we go along.

We've already discussed how ON works. Since H(X) can range anywhere from 0 to 13, and only eight of the possibilities are handled here, any value of H(X) less than 1 or greater than 8 will flow down to 2070 from 2060.

Here values greater than 8 will branch to 3460, and a value of 0 will continue down to the next line, which happens to be 2080. Note that we could have written our ON statement like this:
ON H(X) GOTO 2320,2520,2910,3000,3070,3140,3220,3310,
3460,3460,3460,3460,3460
The computer doesn't mind if it has to go to similar places ON different variables, but in this particular instance it's a lot easier on the programmer to do it the way I've done it rather than as above. But it is not uncommon to see:
ON I GOTO 400,500,500,600
or somesuch in a program when the situation calls for it.

*FIVE-CARD DRAW*

```
2080  DISCARD(X) = 3
    : PRINT B$(10) + STR$ (DI(X))
2090  FOR I = 1 TO 5
2100  FOR II = 1 TO 5
2110  IF V(X,I) < V(X,II) THEN II = 5
    : NEXT II,I
2120  NEXT
2130  K1 = I
    : V(X,K1) = V(X,I)
    : I = 5
    : NEXT
2150  FOR I = 1 TO 5
2155  IF I = K1 THEN 2220
2160  FOR II = 1 TO 5
2180  IF II = K1 THEN 2200
2190  IF V(X,I) < V(X,II) THEN II = 5
    : NEXT II,I
2200  NEXT
2210  K2 = I
    : V(X,K2) = V(X,I)
    : I = 5
2220  NEXT
2250  FOR I = 1 TO 3
2255  IF I > 1 THEN NC(I) = NC(I - 1) + 1
    : GOTO 2270
2260  NC(I) = NC(I) + 1
2270  IF NC(I) = K1 OR NC(I) = K2 THEN 2260

2300  NEXT
2310  GOSUB 20
    : GOSUB 40
    : GOSUB 50
    : GOTO 2820
```

Our first subsegment of the draw routine is for when H(X) = 0. We will have gotten here only with Marvin (player 1) or Betsy (player 4)—anyone else would have dropped out by now. But these hardy souls, ravenous gamblers both, believe that if all else fails, simply hold onto your two highest cards and hope for the best. Marvin will do this all the time; Betsy will do it only if her bankroll is greater than $50. Aside from representing certain

aspects of these players' "personalities," this mulelike stubbornness serves another purpose in the game as a whole. Having players who will stay in on anything guarantees that for much of the game our human player is assured at least one opponent. We don't want to have situations arise regularly where our human player has the only opening hand, upon learning about which all the other players immediately drop out. If our human lasts in the game long enough to beat Marvin and Betsy, he will be facing the cagier Walter and Gerry, who will drop out and let him have their dollar if this situation arises. But in the earlier stages of the game, keeping Marvin and Betsy in guarantees that the game ball will be kept rolling. We're going to make our human work for his winnings.

DISCARD(X) in line 2080 is yet another variable, aka DI(X), which tells us the number of cards our player will be discarding or drawing. The rest of 2080 in this case would print:
I'LL TAKE 3
where 3 is the STR$ value of DI(X). The setup from line 2040 will print this information on the appropriate chat line.

Following this we get some of most convoluted logic this side of the straight-figuring earlier in this chapter. Our problem here is to find the two highest cards, and the only given we have is that no two cards are of like value (otherwise we'd have a pair and we wouldn't be here in the first place). So we set up a pair of loops looking for the highest V values among the five cards; we're going to be looking for I as the highest of the cards. In 2110 we check each card against each other card. If any II value is greater than our I value, we run it again (NEXT II,I). If no II is greater, processing will continue to 2130, where yet another new variable, K1, is initialized with the value of I, and the I loop is finished out. We now know that K1 is the highest card, or more to the point, that V(X,K1) is the highest value. The logic in 2090–2130 may look at first glance as if it won't work: You would think that given the right (or wrong) numbers, you'd sooner or later end up with a NEXT WITHOUT FOR error. But that is not the case. There is no way we'll get to 2120 from 2110. Try plugging in some values and see for yourself. The loops here are perfectly placed; they're just mildly spaghettilike until you get the hang of them.

Lines 2150–2220 recapitulate the search for K1, but here we're looking for K2, the second-highest card, and the only real change is that we make sure we eliminate any chance of finding K1 twice.

So now we know which are our two highest cards, K1 and K2. But how do we use that information? We have to translate these variables into some "universal" variables we can use over and over again in any situation; these variables will be NC(0) through NC(3), which you saw earlier in 2035. In 2250–2300 we translate our knowledge of the two highest cards into a knowledge of the three lowest cards, our discards. In fact, NC(0) is used just for cowcatching; NC(1), NC(2), and NC(3) are the variables we will use to denote the numbers of the cards we will be drawing. Think of NC as New Card; it tells us exactly what we need to know when those cards are drawn. Here's where it happens:

*FIVE-CARD DRAW*

```
20    C$(((X * 5) - 5) + NC(1)) = C$(DR)
    : S$(((X * 5) - 5) + NC(1)) = S$(DR)
    : CD$(((X * 5) - 5) + NC(1)) = CD$(DR)
    : C(((X * 5) - 5) + NC(1)) = C(DR)
    : S(((X * 5) - 5) + NC(1)) = S(DR)
    : DR = DR + 1
    : RETURN
40    C$(((X * 5) - 5) + NC(2)) = C$(DR)
    : S$(((X * 5) - 5) + NC(2)) = S$(DR)
    : CD$(((X * 5) - 5) + NC(2)) = CD$(DR)
    : C(((X * 5) - 5) + NC(2)) = C(DR)
    : S(((X * 5) - 5) + NC(2)) = S(DR)
    : DR = DR + 1
    : RETURN
50    C$(((X * 5) - 5) + NC(3)) = C$(DR)
    : S$(((X * 5) - 5) + NC(3)) = S$(DR)
    : CD$(((X * 5) - 5) + NC(3)) = CD$(DR)
    : C(((X * 5) - 5) + NC(3)) = C(DR)
    : S(((X * 5) - 5) + NC(3)) = S(DR)
    : DR = DR + 1
    : RETURN
```

As we used to say in the sixties, wow! What an incredible trio of program statements. Of course, they are identical replicas of each other, except for the value of X in NC(X). They allow us to replace an old CD$(X) (and an old C, an old S, etc.) with a new value. Look at line 20. Let's say, for instance, that a player is going to draw one card and that that card happens to be card 26 (the value of DR). Let's further say that we are talking about player 1 and that he will be discarding his fourth card—V(1,4). In this case NC(1) would equal 4, the Number of the Card he is discarding. Fill in the blanks. C$(((1*5)−5)+4) equals 4. So C$(4) then becomes C$(DR), and in this case DR = 26, so we have now reinitialized C$(4) as C$(26). Not bad, and we won't need any new programming to know that each player, even after discards, will still hold CD$(1 through 5), or CD$(6 through 10), or whatever.

It is not inconceivable that these three lines 20–50 could have been merged into one, but they are such marvels of detail that it seems best to keep them, and the NCs, separate. But they are placed at the beginning of the program because they are accessed numerous times, and they are very busy. Why not give them the best seat in the house?

We get to these lines from 2310, whence we go to 2820, which we'll cover later. At 2820 we go back to our original hand-figuring subroutine and do it all again, this time with our three new cards. In effect it's as if we have a new hand, and we deal with it accordingly. We'll discuss the details of that when we get to it. Meanwhile, let's proceed while NC(X) is still fresh and confused in your mind.

*FIVE-CARD DRAW*

```
2320  DISCARD(X) = 1
    : PRINT B$(10) + STR$ (DISCARD(X))
2330  NC(1) = 1
2340  FOR I = 3 TO 7
```

```
2350  IF NC(1) = VAL ( MID$ (H$(X),I,1)) THEN
        NC(1) = NC(1) + 1
    : I = 7
    : NEXT
    : GOTO 2340
2360  NEXT
2370  GOSUB 20
2380  IF C(DR - 1) = VAL ( MID$ (H$(X),7)) THEN
        CHECK(X) = V(X,S5(X))
    : H(X) = 9
    : H$(X) = "STRAIGHT TO " + V$(X,S5(X))
    : GOTO 3750
2390  IF IN(X) = 0 THEN 2440
2400  FOR N = 1 TO 5
2410  IF V(X,N) = 1 THEN V(X,N) = 14
2420  NEXT
2430  IN(X) = 0
2440  FOR Y = 1 TO 5
2460  IF Y < > NC(1) AND C(DR - 1) = V(X,Y)
        THEN H$(X) = "PAIR OF " + C$(DR
        - 1) + "S"
2470  NEXT Y
2480  IF LEFT$ (H$(X),1) < > "P" THEN 3750
2500  CHECK(X) = C(DR - 1)
    : H(X) = 6
    : IF CH(X) < 11 THEN H(X) = 5
2510  GOTO 3750
```

This section of programming concerns the possibility of an inside straight. You'll remember that this H$(X) would look like this, ISxxxxx, where the first four little x's tell us the four straight cards in order, and the last X would tell us the V value we would need in the draw card to complete the straight. Here our DISCARD(X) is 1, and we have to find which card from 1 to 5 we need to replace. We start with NC(1) = 1 in 2330, then loop through those first four little X's (or as they're expressed in 2350, MID$(H$(X),I,1)), making sure we don't duplicate one of them. If we do duplicate, we increment NC(1) until we've got it right. Finally, with the correct value of NC(1), we go to 20 for our draw. With our new card in hand, we check in 2380 to see if it is the right value. If so, we set our CHECK(X), then reset H(X) and H$(X), and we're off to 3750, where the screen is reformatted to show the actual draw (we'll describe how that looks later). If not, we reset the IN(X) flag (just in case we've been looking at a low straight) to make sure all our aces in this hand will be worth 14 rather than 1. Then we look for a pair, checking the most recent draw card against the other four in line 2460. Given the way the hands were originally figured, we have only these two possibilities when we draw to an inside straight: Either we get the straight, or we get a pair. Either way, 2480–2510 handle the movement from then on, setting the CHECK(X) and resetting H(X) if necessary and branching in either case to 3750, the screen formatting.

***FIVE-CARD DRAW***

```
2520  DISCARD(X) = 2
    : PRINT B$(10) + STR$ (DI(X))
```

```
2530  FOR I = 1 TO 5
2540  IF ST$(X,I) < > ST$(X,F(X)) THEN NC(1
        ) = I
    : I = 5
2550  NEXT
2560  FOR I = NC(1) + 1 TO 5
2570  IF ST$(X,I) < > ST$(X,F(X)) THEN NC(2
        ) = I
    : I = 5
2580  NEXT
2590  GOSUB 20
    : GOSUB 40
2600  IF S$(DR - 2) < > ST$(X,F(X)) OR S$(D
        R - 1) < > ST$(X,F(X)) THEN 2820
2620  H$(X) = "FLUSH " + S$(DR - 1) + "S"
2630  REM   CHECK FOR STRAIGHT FLUSH
2635  IN(X) = 0
2640  FOR Z1 = 1 TO 5
2641  FOR Z5 = 1 TO 5
2642  IF V(X,Z5) = V(X,Z1) + 4 THEN S1 = Z1

    : S5 = Z5
    : Z1 = 5
    : Z5 = 5
    : NEXT Z5,Z1
    : GOTO 2649
2643  NEXT Z5,Z1
2644  IF IN(X) = 1 GOSUB 2740
    : GOTO 3750
2645  FOR I = 1 TO 5
2646  IF V(X,I) = 14 THEN V(X,I) = 1
2647  NEXT
2648  GOTO 2640
2649  FOR Z2 = 1 TO 5
2650  IF V(X,Z2) = V(X,S1) + 1 THEN S2 = Z2

    : Z2 = 5
    : NEXT
    : GOTO 2653
2651  NEXT
2652  GOSUB 2740
    : GOTO 3750
2653  FOR Z3 = 1 TO 5
2654  IF V(X,Z3) = V(X,S2) + 1 THEN S3 = Z3

    : Z3 = 5
    : NEXT
    : GOTO 2657
2655  NEXT
2656  GOSUB 2740
    : GOTO 3750
2657  FOR Z4 = 1 TO 5
2658  IF V(X,Z4) = V(X,S3) + 1 THEN H$(X) =
        "STRAIGHT FLUSH"
    : Z4 = 5
    : NEXT
    : H(X) = 13
    : GOTO 3750
```

```
2659  NEXT
2730  GOSUB 2740
    : GOTO 3750
2740  IN(X) = 0
2741  FOR I = 1 TO 5
2742  IF V(X,I) = 1 THEN V(X,I) = 14
2743  NEXT
2745  FOR I = 1 TO 5
2750  FOR N = 1 TO 5
2760  IF V(X,I) < V(X,N) THEN N = 5
    : NEXT N,I
    : IF I = 6 THEN CHECK(X) = V(X,5)
    : GOTO 2800
2770  NEXT
2780  CHECK(X) = V(X,I)
    : I = 5
2790  NEXT
2800  H(X) = 10
    : RETURN
```

The programming above handles our three-flush situation. Two cards are discarded, and in 2540 and 2570 we get the two cards that are not of the correct flush suit (ST$(X,F(X))), which we set back in the hand-figuring routine. If either of these new draw cards is not in the correct suit, we branch to 2820, whence the whole hand will be refigured with the new cards. The reason for this is that anything from a straight to three of a kind could possibly have occurred with that draw of two cards, and we want to cover all the possibilities, unlike the nice, neat leftover pair possibility after we had eliminated our inside straight earlier.

If we do have a flush, we so note it in 2620, and then we have a lot more business to take care of. First we have to see if in fact we have a straight flush, which is not at all impossible considering the draw of two cards. Lines 2630–2659 handle this possibility for us. If at any time we fall out of the possibility, we branch to 2740, where we reset any aces if necessary, then get our CHECK(X) and H(X) values. And then we go to 3750 for screen formatting. We don't pull any CHECK(X) if we do have the straight flush. The odds of this happening are remote but possible, so we're ready for that; the odds of *two* people getting a straight flush in one hand are so insignificant that they aren't even worth bothering about.

***FIVE-CARD DRAW***

```
2820  REDO = 1
    : GOSUB 100
    : REDO = 0
2840  IF H(X) = 6 THEN CHECK(X) = V(X,
        VAL ( MID$ (H$(X),2,1)))
    : H$(X) = "PAIR OF " + V$(X, VAL (
        MID$ (H$(X),2,1))) + "S"
2850  IF H(X) = 7 AND VAL ( MID$ (H$(X),3,
        1)) > VAL ( MID$ (H$(X),5,1))
        THEN CHECK(X) = V(X, VAL ( MID$ (H
        $(X),3,1)))
    : H$(X) = "TWO PAIRS "
```

```
2860  IF H(X) = 7 AND VAL ( MID$ (H$(X),3,
        1)) < VAL ( MID$ (H$(X),5,1))
        THEN CHECK(X) = V(X, VAL ( MID$ (H
        $(X),5,1)))
    : H$(X) = "TWO PAIRS"
2870  IF H(X) = 8 THEN CHECK(X) = V(X,
        VAL ( RIGHT$ (H$(X),1)))
    : H$(X) = "THREE " + V$(X, VAL (
        RIGHT$ (H$(X),1))) + "S"
2880  IF H(X) = 9 THEN CHECK(X) = V(X,F3)
2890  IF H(X) = 11 OR H(X) = 12 THEN
        GOSUB 32000
2900  GOTO 3750
```

This is the part of the program from which we branch back to the original hand-figuring. First we set a new variable flag, REDO, at 1, then we GOSUB 100 (not 90, the actual beginning of the subroutine, because that line begins the FOR X = 1 TO 5 loop, and here we're interested only in one particular already-defined X). You'll remember encountering REDO in that earlier part of the program. By setting this flag we ensure that we will return to this part of the program without having to go through unnecessary parts of the earlier subroutine. What we've done by using this flag this way is turn one subroutine into two subroutines, using the same lines of programming for the same ends but from different starting and finishing points. (That multiple usage is what subroutines are all about.)

There is another way of handling flags, as long as the flag has only two possible conditions, either on or off. You'll remember our discussion of Boolean logic earlier, and how the computer has its own sense of seeking truth. If so directed, the computer can evaluate the truth of pure numeric expressions without equal signs or other operators. To the computer, a value of 0 is false, a value of 1 is true. So look back at line 1520:

1520 IF REDO = 1 THEN RETURN

This could have been more briefly expressed as:

1520 IF REDO THEN RETURN

The value of REDO is 1. If we simply evaluate our flag as in the revised 1520, and that value turns out to be 1, or truth, the condition is met and the RETURN is carried out. The statement IF NOT REDO would be the equivalent of IF REDO = O. I find it easier to use numbers, and be warned that the use of Boolean logic in this fashion, with NOTs and no NOTs, can be very tricky, so handle with care. One thing, though: Their use is very impressive in a program and really makes you look as though you know what you're doing. End digression.

Lines 2840–2890 handle the H$(X) names that wouldn't have been derived from the hand-figuring begun at 100, as well as putting their CHECK(X) values on them. The 32000 GOSUB is more of the same and looks like this:

*FIVE-CARD DRAW

```
32000 REM  FULL HOUSE AND FOUR OF A KIND C
        HECK(X) ROUTINE
32010 CH$ = RIGHT$ (H$(X),2)
32020 IF CH$ = "ES" THEN CH(X) = 14
    : GOTO 32140
32030 IF CH$ = "GS" THEN CH(X) = 13
    : GOTO 32140
32040 IF CH$ = "NS" THEN CH(X) = 12
    : GOTO 32140
32050 IF CH$ = "KS" THEN CH(X) = 11
    : GOTO 32140
32060 IF CH$ = "OS" THEN CH(X) = 10
    : GOTO 32140
32070 CH(X) = VAL ( MID$ (H$(X), LEN (H$(X
        )) - 1,1))
32140 RETURN
```

Here we read the ends of the H$(X) names (acES, kinGS, etc.) to derive the CHECK(X). The CH$(X) is merely a one-time use variable for this operation.

*FIVE-CARD DRAW*

```
2910  DISCARD(X) = 1
    : PRINT B$(10) + STR$ (DI(X))
2920  NC(1) = 1
2930  FOR I = 3 TO 7
2940  IF NC(1) = VAL ( MID$ (H$(X),I,1)) THEN
        NC(1) = NC(1) + 1
    : I = 7
    : NEXT
    : GOTO 2930
2950  NEXT
2960  GOSUB 20
2970  IF C(DR - 1) = V(X, VAL ( MID$ (H$(X)
        ,3,1))) - 1 THEN CHECK(X) = V(X, VAL
        ( MID$ (H$(X),6,1)))
    : H$(X) = "STRAIGHT TO " + V$(X, VAL ( MID$
        (H$(X),6,1)))
    : H(X) = 9
    : GOTO 3750
2980  IF C(DR - 1) = V(X, VAL ( MID$ (H$(X)
        ,6,1))) + 1 THEN H$(X) = "STRAIGHT
        TO " + C$(DR - 1)
    : CHECK(X) = C(DR - 1)
    : H(X) = 9
    : GOTO 3750
2985  IF C(DR - 1) = 14 AND VAL ( MID$ (H$(
        X),3,1)) = 2 THEN C(DR - 1) = 1
    : IN(X) = 1
    : GOTO 2970
2990  V(X,NC(1)) = C(DR - 1)
    : GOTO 2390
```

Our next possible hand is the four straight, with an H$(X) of "4Sxxxx", the little X's the numbers of the cards in the player's hand, in order, that contain the straight values. After getting our NC(I) as before through augmentation in 2940, we plug in the C value of DR − 1, our draw card, in lines 2970–2980. We do it top and bottom, since the straight could be hit either way. Line 2985 looks for a low ace for a 1-to-5 straight. If we get our straight, fine, if not, back to 2390, where IN(X) is reset and we look for the pair, our only possibility at this point (there's no way of having a flush; that possibility was eliminated in the original hand-figuring where a four flush would have taken precedence over a four straight).

*FIVE-CARD DRAW*

```
3000  DISCARD(X) = 1
    : PRINT B$(10) + STR$ (DI(X))
3010  FOR I = 1 TO 5
3020  IF ST$(X,I) < > ST$(X,F(X)) THEN NC(1
        ) = I
    : I = 5
3030  NEXT
3040  GOSUB 20
3050  IF S$(DR - 1) = ST$(X,F(X)) THEN 2620

3060  GOTO 2820
```

Here we're looking for the match to our four flush. If we get it, then on to 2620 to check for the straight flush, etc. If not, it's up to 2820 and a REDO. We could have a pair, and we could even have a straight. A full REDO will handle both for us.

*FIVE-CARD DRAW*

```
3070  DISCARD(X) = 3
    : PRINT B$(10) + STR$ (DI(X))
3080  FOR I = 1 TO 3
3090  NC(I) = NC(I - 1) + 1
3100  IF NC(I) = VAL ( MID$ (H$(X),2,1)) OR
        NC(I) = VAL ( MID$ (H$(X),3,1))
        THEN NC(I) = NC(I) + 1
    : GOTO 3100
3110  NEXT
3120  GOSUB 20
    : GOSUB 40
    : GOSUB 50
    : GOTO 2820
```

Lines 3070–3120 are for when we have an H(X) of 5, a small pair. We find our NCs in the usual way and once again do a whole REDO. The next segment handles jacks or better:

*FIVE-CARD DRAW*

```
3140  IF LEN (H$(X)) = 4 THEN 3070
```

```
3150  DISCARD(X) = 2
    : PRINT B$(10) + STR$ (DI(X))
3160  FOR I = 1 TO 2
3170  NC(I) = NC(I - 1) + 1
3180  IF NC(I) = VAL ( MID$ (H$(X),2,1)) OR
        NC(I) = VAL ( MID$ (H$(X),3,1))
        OR NC(I) = VAL ( RIGHT$ (H$(X),1)) THEN
        NC(I) = NC(I) + 1
    : GOTO 3180
3190  NEXT
3200  GOSUB 20
    : GOSUB 40
    : GOTO 2820
```

You'll remember here that our H$(X) in this situation will look like "PxxJ"—with one exception. If Walter is keeping a kicker it will look like "PxxJKx"—with a LEN(H$(X)) of 6. Hence line 3140. If we have a regular pair of jacks or better, it can be handled just as it was earlier. But if Walter is keeping the kicker, then his DISCARD(X) is 2 and we proceed accordingly. But once again we go the REDO route, in any case.

*FIVE-CARD DRAW*

```
3220  PRINT B$(10) + "1"
    : DI(X) = 1
    : NC(1) = 1
3240  FOR I = 3 TO 7
3250  IF NC(1) = VAL ( MID$ (H$(X),I,1)) THEN
        NC(1) = NC(1) + 1
    : I = 7
    : NEXT
    : GOTO 3240
3260  NEXT
3270  GOSUB 20
3280  IF C(DR - 1) = V(X, VAL ( MID$ (H$(X)
        ,3,1))) THEN H$(X) = "FULL HOUSE, "
        + C$(DR - 1) + "S"
    : CHECK(X) = C(DR - 1)
    : H(X) = 11
    : GOTO 3750
3290  IF C(DR - 1) = V(X, VAL ( MID$ (H$(X)
        ,5,1))) THEN H$(X) = "FULL HOUSE, "
        + C$(DR - 1) + "S"
    : CHECK(X) = C(DR - 1)
    : H(X) = 11
    : GOTO 3750
3300  GOTO 2850
```

As you can see, we're speeding up our discussion now, since much of this programming is repetitious. In fact, much of *all* programming is repetitious, and it's the ability to overcome the tedium and keep going that separates the hackers from the hacks. In a way, what we're doing now is much like the conversation with the computer segment in "Space Derelict!," the same thing over and over again, only with different values plugged in. A fact of

life, I'm afraid, but bear with it a little while longer. There are still a few tricks up our sleeves that are yet to be unveiled.

Lines 3220–3300 handle our draw to two pairs. We don't have to REDO here because either our draw matches one of our pairs for a full house or it doesn't. If we have only the pairs, we go to 2850 in 3300 to get our H$(X) set up.

***FIVE-CARD DRAW***

```
3310  IF LEN (H$(X)) > 4 THEN PRINT B$(10)
        + "1"
    : DI(X) = 1
3320  IF LEN (H$(X)) = 4 THEN PRINT B$(10)
        + "2"
    : DI(X) = 2
3340  FOR I = 1 TO 2
3350  NC(I) = NC(I - 1) + 1
3360  IF NC(I) = VAL ( MID$ (H$(X),2,1))
        OR NC(I) = VAL ( MID$ (H$(X),3,1))
        OR NC(I) = VAL ( MID$ (H$(X),4,1))
        THEN NC(I) = NC(I) + 1
    : GOTO 3360
3370  NEXT
3380  GOSUB 20
    : IF LEN (H$(X)) > 4 THEN 3430
3390  GOSUB 40
3400  IF C(DR - 1) = C(DR - 2) THEN H$(X)
        = "FULL HOUSE, " + V$(X, VAL (
        RIGHT$ (H$(X),1))) + "S"
    : H(X) = 11
    : GOSUB 32000
3410  IF C(DR - 1) = V(X, VAL ( RIGHT$ (H$
        (X),1))) OR C(DR - 2) = V(X, VAL (
        RIGHT$ (H$(X),1))) THEN H$(X) = "F
        OUR " + V$(X, VAL ( RIGHT$ (H$(X),
        1))) + "S"
    : GOSUB 32000
3420  IF LEFT$ (H$(X),1) = "T" THEN CHECK(
        X) = V(X, VAL ( MID$ (H$(X),2,1)))

    : H$(X) = "THREE " + V$(X, VAL ( MID$
        (H$(X),2,1))) + "S"
3425  GOTO 3750
3430  IF C(DR - 1) = 14 THEN H$(X) = "FULL
         HOUSE, " + V$(X, VAL ( MID$ (H$(X
        ),2,1))) + "S"
    : GOSUB 32000
3440  IF C(DR - 1) = V(X, VAL ( MID$ (H$(X
        ),2,1))) THEN H$(X) = "FOUR " + V$
        (X, VAL ( MID$ (H$(X),2,1)))
    : GOSUB 32000
3450  GOTO 3420
```

Remember Walter and his kicker? Well, he might have done it again with three of a kind; hence lines 3310–3320. All our H$(X) naming is handled

in this segment, with the appropriate branching for Walter and NOT Walter (which is *not* Boolean logic, which is why I tend to steer clear of it). And finally:

*FIVE-CARD DRAW*

```
3460  PRINT "I'LL PLAY THESE." + G$
    : GOSUB 63000
    : GOSUB 27180
    : GOTO 4190
```

If our computer player has a pat hand and does not need to draw, we get his CHECK(X) at 27180, and we go to 4190. We don't have to format the screen for this one; if computers could grin, this is when they'd be doing it. And thus ends our search for draw cards for the computer-generated players.

## DRAWING THE CARDS ON THE SCREEN

This is a very short routine, and we'll skip down to it now before handling the human player's draw, since this flows more logically from what we've just been talking about. The problem we have here is that each player has said how many cards he wants, and that is what our human player has seen on the screen in the appropriate chat lines. We've also done the draw interally and refigured the hands, which is neither here nor there to our human player. Now we have to put in a little drama . . . we have to make it look as though our players are actually drawing cards. So we're going to erase the appropriate number of card backs, those seven inverse X's, and redraw them on the screen. And now that we have screen formatting down to a science, we can do it most efficiently.

*FIVE-CARD DRAW*

```
3730  PRINT B$(12) + G$
    : RETURN
3740  INVERSE
    : PRINT B$(0) + G$
    : NORMAL
    : RETURN
3750  FOR I = DI(X) - 1 TO 0 STEP - 1
3770  VTAB 6 - I
    : IF X > 2 THEN VTAB 14 - I
3775  HTAB 1
    : IF X / 2 = INT (X / 2) THEN HTAB 20
3780  GOSUB 3730
3790  GOSUB 63020
3800  NEXT
3810  FOR I = DI(X) - 1 TO 0 STEP - 1
3820  VTAB 6 - I
    : IF X > 2 THEN VTAB 14 - I
3825  HTAB 1
    : IF X / 2 = INT (X / 2) THEN HTAB 20
```

```
3830  GOSUB 3740
3840  GOSUB 63020
3850  NEXT
```

This routine starts at 3750, to which we would have branched from the draw routine. Lines 3730–3740 are a pair of minisubroutines just waiting to happen, with no attempt made at speed because we want the formatting to go slowly: It makes the act of taking the draw cards that much more realistic. So in 3750 we set a backward loop from DISCARD(X)/DI(X) − 1 To 0, STEP −1; if the DI(X) was 3, then we'll go 2, 1, 0; if DI(X) was 2, then we'll go 1, 0; if it was 0, then we'll just have the one step, 0. In this loop we're making sure we count the number of discards accurately. The reason we do it this way, starting at I = DI(X) − 1, is because we're going to be using the I value to help position our cursor. Also, even though we know hypothetically that FOR . . . NEXT loops can go either up or down (i.e., positive or negative STEP values), it's nice to see the more uncommon down direction in action.

We set our VTAB in 3770 as 6 − I for players 1 and 2; we set it at 14 − I for players 3 and 4. In 3775 we set HTAB 1 for players 1 and 3 and at HTAB 20 for the even-numbered players, 2 and 4: We've pulled this same INT trick in both our anteing and betting routines. The logic in these two lines is dependent on an IF . . . THEN command, and it points out what I feel is one of the few flaws of Applesoft, the lack of an IF . . . THEN . . . ELSE command. A lot of dialects of Basic do use the ELSE, which can make life a lot easier in situations like this. What we would have said was:

IF X < 3 THEN VTAB 6 − I ELSE VTAB 14 − I

which is certainly much neater than the way we're forced to express it. But that's life.

Now that we've placed our cursor, we do the GOSUB at 3730, which erases our seven inverse X's and rings the bell for up to three discards. Then we replace the cursor back at the first discard line, and this time we print a *new* set of seven inverse X's in 3740. We haven't done anything, really, but it looks as though we have, and sometimes that's just as important.

## THE HUMAN GETS HIS CARDS

Needless to say, we handle the human draw quite differently from the computer ones. Here the choice of what to do is entirely up to the player, and all we have to do is provide clear directions to the player. Our problem is that, given the tightness of the screen format, there's nowhere for these directions to go, so we have to engineer things just right so the player doesn't curse us for ruining his game.

***FIVE-CARD DRAW***

```
3470  INPUT "HOW MANY CARDS?";DI(5)
3480  GOSUB 7160
3490  IF DI(5) < > INT (DI(5)) OR DI(5)
        > 3 OR DI(5) < 0 THEN GOSUB 7160
    : GOTO 3470
```

```
3500  IF DI(5) = 0 THEN PRINT "OKAY."
    : GOTO 4190
3510  I = DI(5)
3520  FOR VT = 18 TO 22
3530  VTAB VT
    : HTAB 1
    : INVERSE
    : FLASH
    : PRINT CD$(VT + 3)
    : NORMAL
3540  GOSUB 7180
    : CALL - 868
    : GOSUB 7180
3550  INPUT "REPLACE CARD (Y/N)?";AN$
3560  IF AN$ < > "Y" AND AN$ < > "N"
        THEN 3540
3570  IF AN$ = "N" THEN VTAB VT
    : HTAB 1
    : INVERSE
    : PRINT CD$(VT + 3)
    : NORMAL
    : GOTO 3640
3580  VTAB VT
    : HTAB 1
    : PRINT B$(12)
3590  VTAB VT
    : HTAB 1
    : INVERSE
    : PRINT B$(0)
    : NORMAL
3600  CD$(VT + 3) = CD$(DR)
    : C$(VT + 3) = C$(DR)
    : S$(VT + 3) = S$(DR)
    : C(VT + 3) = C(DR)
    : S(VT + 3) = S(DR)
3610  MAN(DI(5)) = VT
    : REM   TO KEEP PLAYER FROM CHEATING ON
        DRAW
3620  DR = DR + 1
3630  DI(5) = DI(5) - 1
    : IF DI(5) = 0 THEN VT = 22
3640  NEXT
3650  IF DI(5) < > 0 THEN 3520
3660  GOSUB 7180
    : CALL - 868
    : GOSUB 63000
3680  FOR N = 1 TO I
3690  VTAB (MAN(N))
    : HTAB 1
    : PRINT B$(12)
3700  VTAB (MAN(N))
    : HTAB 1
    : INVERSE
    : PRINT CD$(MAN(N) + 3) + G$
    : GOSUB 63020
    : NORMAL
```

```
3710  NEXT
3720  GOTO 4190
```

The first thing we do is get the input of DI(5) from the player in line 3470. This will, of course, appear on the human's chat line. Line 3490 cowcatches bad inputs and sends us back to 3470, while 3500 simply takes us out of here if the human stands pat. But if the human does indicate a desire to draw cards, we are going to go down the line of his cards and FLASH each of them one at a time, asking him if he wants to replace this particular card. If "Y", we will replace this FLASHer; if "N", we will go on to the next card. The CALL -868 in 3540 clears one complete line from the cursor horizontally to the end of the screen; because of the length of the text on the human's chat line, our B$(12) string eraser simply won't do the job for us. This is a nice CALL to remember; it comes in handy at times like these.

The loop here stretches from 3520 to 3640, and the VTAB/HTAB part of it should be clear to you by now, as should the plugging in of the new values in 3600. What we do that's tricky is glimpsed in line 3610. We don't just automatically print in the CD$ of the new card. That would mean that the player would see each new card as he draws it, possibly allowing him to change his draw in midstream. So we circumvent this by printing in B$(0), the seven inverse X's, one set at a time until the player has taken all his cards, DI(5) (line 3630). If we go through the loop and the player hasn't taken enough cards, he'll go through it again, thanks to 3650. When we're finally done, we bring into play our MAN variable in 3680–3710. We've already set I as the original DI(5); now we print in the new CD$s one at a time, where they're supposed to go (thanks to MAN(N)), and that way nobody cheats and the computer is guaranteed a fair game.

Last but not least in the draw segment comes this:

***FIVE-CARD DRAW***

```
4190  X = X + 1
4200  IF X = 6 THEN X = 1
4210  IF X = 1 AND D < > 5 THEN 2030
4220  IF X < > 1 AND X < > D + 1 THEN 2030
4230  RETURN
```

This merely augments our X values appropriately to keep the hand moving from one player to the next. Lines 4210–4220 keep us going if there are more players to come, and 4230 RETURNs us if the draw segment is over.

## SETTING THE NEW PATTERNS OF BETTING

Our next subroutine, as indicated by our line 10000 traffic director, begins at line 5000. Here we will update the various BF(X), DF(X), and RF(X)

variables based on the new hands held by the computer players. This section is identical in structure to the earlier version back at line 1570, but the values of the variables will be radically different from the first time around. The most dramatic difference is the addition of bluffing elements.

*FIVE-CARD DRAW*

```
5000  FOR X = 1 TO 4
5010  IF P(X) < > 0 THEN 5030
5020  ON X GOSUB 5050,5110,5190,5270
5030  NEXT
5040  RETURN
```

The introduction to this section is written in classic Basic format. We set our loop in 5000, pull out our exceptions in 5010, do an ON . . . GOSUB in 5020, a NEXT in 5030, and a RETURN in 5040. Nothing could be simpler to understand, and nothing could so easily show the concept of subroutine in its best light. Here we have boiled down our processing to its essential ingredients: We don't fool around with REMs or complicated, convoluted logic, we don't do any processing of redundant or unimportant data. We do just what we're supposed to do and then RETURN. Would that all logic and programming were so simple.

*FIVE-CARD DRAW*

```
5050  REM  MARVIN
5060  IF H(1) < 6 THEN DF(1) = 0
    : BF(1) = 0
5070  IF H(1) < 8 AND H(1) > 5 THEN BF(1) =
        1
    : RF(1) = 0
    : DF(1) = 14
5080  IF H(1) > 7 AND H(1) < 10 THEN BF(1) =
        2
    : RF(1) = 2
    : DF(1) = 21
5090  IF H(1) > 9 THEN BF(1) = 3
    : RF(1) = 5
    : DF(1) = 21
5100  RETURN
```

Good old Marvin hasn't changed much now that he's taken his draw. Of course, now he doesn't stay in on anything—if H(1) is less than jacks or better, he'll drop out (5060)—but the same uncomplicated, blunt style of betting is still readily apparent. Note that here and following we must be sure to set BF(X) as 0 (line 5060) if our player has a bum hand. Otherwise we might have a lingering BF(X) from the first round of betting, which would keep our player in the game when he's not supposed to be there.

*FIVE-CARD DRAW*

```
5110  REM  GERRY
```

```
5120  IF H(2) = 3 OR H(2) = 4 THEN BL(X) =
        INT ( RND (1) * 10)
    : IF BL(X) < 2 THEN BF(2) = 2
    : RF(2) = 3
    : DF(2) = 14
    : RETURN
5130  IF H(2) < 6 THEN DF(2) = 0
    : BF(2) = 0
5140  IF H(2) = 6 OR H(2) = 7 THEN BF(2) =
        1
    : RF(2) = 0
    : DF(2) = 8
5150  IF H(2) = 8 OR H(2) = 9 THEN BF(2) =
        2
    : RF(2) = 1
    : DF(2) = 12
    : IF H(2) = 9 THEN DF(2) = 21
5160  IF H(2) = 10 THEN BF(2) = 3
    : RF(2) = 3
    : DF(2) = 21
5170  IF H(2) > 10 THEN BF(2) = 3
    : RF(2) = 5
    : DF(2) = 21
5180  RETURN
```

Gerry, on the other hand, now introduces the possibility of bluffing with a bum hand of H(2) = 3 or H(2) = 4 (incompleted four straight and four flush, respectively). We do this in 5120 by rolling a random number for the new variable BL(X), as in BLuff. We arbitrarily give Gerry two chances out of 10 (INT(RND(1)*10) gives us numbers from 0 to 9) to run a bluff and then assign betting variables to him if he hits one of those two chances. Gerry's bluffs are going to be rare, but they still will be a distinct possibility—which is the way any good player approaches draw poker. And he will be conservative about them, playing up to only $14 on his DF. He's not out to cut his own throat in these last-ditch bluff attempts—he just wants you, the human, to drop out.

Without the bluffing, Gerry's betting proceeds much as it did before, a tad cautious but nonetheless reasonable. In line 5150 you see in the last statement how we manage to give two different values to DF(2) for the two different H(2)s without having to break out of the programming line. Remember, the more we can squeeze behind one line number, the better.

***FIVE-CARD DRAW***

```
5190  REM  WALTER
5200  IF H(3) < 3 THEN DF(3) = 0
    : BF(3) = 0
5210  IF H(3) = 3 OR H(3) = 4 THEN BL(X) =
        INT ( RND (1) * 10)
    : IF BL(X) = 7 THEN BF(3) = 2
    : RF(3) = 3
    : DF(3) = 10
```

```
5220  IF BL(X) < > 7 AND H(3) < 6 THEN DF(3
        ) = 0
    : BF(3) = 0
5230  IF H(3) < 8 AND H(3) > 5 THEN BF(3) =
        1
    : RF(3) = 0
    : DF(3) = 7
5240  IF H(3) = 8 THEN BF(3) = 2
    : RF(3) = 2
    : DF(3) = 21
5250  IF H(3) > = 9 THEN BF(3) = 3
    : RF(3) = 3
    : DF(3) = 21
5260  RETURN
```

Walter too will bluff, but he does it all wrong. He does it once in a blue moon (line 5210), and then, given his DF(3) of 10, he hardly sticks with it. It's as if he makes his bet as though he had filled the straight or flush, then thinks better of it and drops out. If our human player can spot this trait it will make Walter that much more "human."

**FIVE-CARD DRAW***

```
5270  REM  BETSY
5280  IF H(4) < 9 THEN BL(X) = INT ( RND (
        1) * 4) + 1
    : IF BL(X) = 4 THEN BF(4) = 3
    : RF(4) = 3
    : DF(4) = 21
    : GOTO 5340
5290  IF H(4) < 6 THEN DF(4) = 0
    : BF(4) = 0
5300  IF H(4) = 6 THEN BF(4) = 2
    : RF(4) = 0
    : DF(4) = 8
5310  IF H(4) = 7 THEN BF(4) = 2
    : RF(4) = 1
    : DF(4) = 10
5320  IF H(4) = 8 THEN BF(4) = 2
    : RF(4) = 2
    : DF(4) = 12
5330  IF H(4) > 8 THEN BF(4) = 3
    : RF(4) = 3
    : DF(4) = 21
5340  IF R(4) > 50 AND DF(4) > 0 THEN BF(4
        ) = BF(4) + 1
    : RF(4) = RF(4) + 1
    : IF R(4) > 100 THEN BF(4) = BF(4) + 1

    : RF(4) = RF(4) + 1
    : DF(4) = DF(4) + 1
5350  RETURN
```

On the other hand, Betsy bluffs like a house afire—one time out of four! And she bluffs with any hand from H(4) = 0 to H(4) = 8 (two pairs). Very unpredictable and even a little self-destructive, but you've got to give her

credit for guts. To complicate things further, she also keeps a constant eye on her stack of chips. Take a look at 5340. If her bankroll is greater than $50, she'll bet even more heavily. And if it's greater than $100, more heavily still (note that all her RFs and BFs are designed so she won't bet past the limit). A difficult opponent to read at any time, and for the human, in some respects as hard to beat as Gerry, provided the cards go right for her. If they go wrong, she'll burn herself out. Again, this fluidity makes her that much more human in the eyes of the real human player.

## THE SECOND ROUND OF BETTING

This subroutine is designed much the same as the first round of betting. Unfortunately, we cannot use that first subroutine a second time because originally we had the very different factor of opening the bidding, which we don't have here. But nonetheless some of these lines are directly copied, and they were copied by the computer itself. Here's how.

Way back in the beginning of this book we discussed using the Escape I, J, K, and M keys for making cursor moves when editing a program (actually, I referred you to the manuals for this truly fundamental concept—no fool, I). Well, one of the things we can also edit in a line of programming is the line number itself. Let's look at lines 8420 and 9140 (don't worry about typing these two in now):

```
8420  IF B(OP) > R(5) OR B(OP) > BE - LB(OP
        ) + 5 OR B(OP) < 0 OR (B(OP) > BE -
        LB(OP) AND NB = 3) OR B(OP) < > INT
        (B(OP)) THEN GOSUB 7160
    : PRINT "ILLEGAL BET" + G$
    : GOSUB 63000
    : GOSUB 7160
    : GOTO 8400
9140  IF B(OP) > R(5) OR B(OP) > BE - LB(OP
        ) + 5 OR B(OP) < 0 OR (B(OP) > BE -
        LB(OP) AND NB = 3) OR B(OP) < > INT
        (B(OP)) THEN GOSUB 7160
    : PRINT "ILLEGAL BET" + G$
    : GOSUB 63000
    : GOSUB 7160
    : GOTO 9120
```

Except for the final GOTO, they are identical. First I typed in all of the original 8420. Then, when I needed that exact same, very long line later, I simply typed:
LIST 8420
and all of line 8420 appeared on the monitor. Unfortunately, it spread out all over the screen, with spaces put in by the computer that I didn't want, so then I typed in:

POKE 33, 33

This POKE eliminates all unnecessary spaces from a programming line

statement while keeping in all the necessary spaces (POKE 33, 33 resets the text window seven spaces in from the right, but it has this extra formatting benefit built into it). After doing this POKE I retyped:
LIST 8420
and this time it came out just the way I wanted. So I pressed Escape, and then, using the I, J, K, and M keys, positioned the cursor at the beginning of the line. Then I typed in the new number, 9140, over the old number, 8420. This done, and now using the arrow key (the one that points to the right), I simply passed the cursor through all the rest of the line, changing that last GOTO when I got to it. When I hit Return, I had re-created the spit and image of 8420, resurrected as 9140, *without any change in the original 8420*. Then I typed in TEXT, which resets the text window (POKE 33, 40 would do the same), and that was that. If you know this trick, fine; if you don't, experiment for a second. It's a good one to have in your own little bag.

*FIVE-CARD DRAW*

```
9000  REM  SECOND ROUND OF BETTING
9010  POKE 49168,0
9020  FOR OP = 1 TO 5
    : LB(OP) = 0
    : NEXT
9050  OP = STARTER
    : RD = 1
    : NB = 0
9060  GOTO 9080
9070  OP = OP + 1
    : IF OP = 6 THEN OP = 1
9080  IF P(OP) < > 0 THEN RD = RD + 1
    : GOTO 9260
9090  GOSUB 31000
9100  ON OP GOSUB 7000,7040,7080,7120,7160
9110  IF OP < > 5 THEN 9200
```

Once again we clear a few variables and initialize a few others. OP = STARTER is in 9050 because the second round of betting begins with the original opener, not the player to the left of the dealer. And we move the play along through each of the five players, incrementing the RD round as necessary. Line 9250 (from 9080) is the continuation of the incrementation process.

*FIVE-CARD DRAW*

```
9120  INPUT "WHAT'S YOUR BET?";B(OP)
9130  GOSUB 7180
    : PRINT G$
9140  IF B(OP) > R(5) OR B(OP) > BE - LB(OP
      ) + 5 OR B(OP) < 0 OR (B(OP) > BE -
      LB(OP) AND NB = 3) OR B(OP) < > INT
      (B(OP)) THEN GOSUB 7160
    : PRINT "ILLEGAL BET" + G$
    : GOSUB 63000
    : GOSUB 7160
    : GOTO 9120
```

```
9150  IF B(OP) < > 0 AND B(OP) < BE - LB(OP
        ) THEN GOSUB 7160
    : PRINT "ILLEGAL BET" + G$
    : GOSUB 63000
    : GOSUB 7160
    : GOTO 9120
9160  IF B(OP) = 0 AND BE > 0 THEN P(5)
        = 10
    : GOSUB 40000
9170  R(5) = R(5) - B(OP)
    : PT = PT + B(OP)
    : RD = RD + 1
    : IF BE = 0 THEN BE = B(OP)
    : RD = 1
9180  IF B(OP) > BE - LB(OP) THEN RF(OP) =
        B(OP) - BE - LB(OP)
    : BE = BE + RF(OP)
    : IF BE < > 0 THEN NB = NB + 1
    : RD = 1
9190  LB(OP) = BE
    : GOTO 9250
```

Here our human bets his hand. In 9140 and 9150 he's up against all the restrictions as before. Note that if his LB(OP) = 0 it doesn't affect these lines; in fact, they are playable whether or not he or any other player has already bet in this round. That was to save ourselves some of the longer programming we had in the first round of betting. This time we can work it that much cleaner because we don't have that opening factor.

*FIVE-CARD DRAW*

```
9200  IF BF(OP) > 0 AND NB = 0 AND BE = 0 THEN
        PRINT B$(4) + STR$ (BF(OP)) + G$
    : LB(OP) = BF(OP)
    : BE = BF(OP)
    : RD = 1
    : PT = PT + BE
    : R(OP) = R(OP) - BE
    : GOTO 9250
9210  IF DF(OP) < BE THEN PRINT B$(9) + G$
    : RD = RD + 1
    : P(OP) = 10
    : GOSUB 40000
    : GOTO 9250
9220  IF NB < 3 AND RF(OP) > 0 THEN
        PRINT B$(3) + STR$ (RF(OP)) + G$
    : R(OP) = R(OP) - RF(OP) - BE + LB(OP)
    : RD = 1
9230  IF NB < 3 AND RF(OP) > 0 THEN PT = PT
        + RF(OP) + BE - LB(OP)
    : BE = BE + RF(OP)
    : LB(OP) = BE
    : NB = NB + 1
    : GOTO 9250
```

```
9240  LB(OP) = BE - LB(OP)
    : RD = RD + 1
    : PT = PT + LB(OP)
    : R(OP) = R(OP) - LB(OP)
    : PRINT B$(2) + G$
```

The same holds true for our computer players. The programming here is very tight; line 9200, for instance, seems to have about 9200 variables in it. But they're all there where they're supposed to be, raising this value, lowering that one, initializing the other. Since we've already explained what's going on here, we won't go into detail now.

*FIVE-CARD DRAW*

```
9250  GOSUB 63000
9260  IF RD < > 5 THEN 9070
9270  BE = 0
    : LB(OP) = 0
    : GOSUB 31000
9280  REM  CHECK FOR DEFAULT WINNER
9300  FOR N = 1 TO 5
9305  IF P(N) < > 0 THEN SOFAR = SOFAR + 1

9310  NEXT
9320  IF SOFAR < > 4 THEN RETURN
9330  GOTO 8600
```

Finally we look for our default winner using the same SOFAR variable from last time. If we have one, we go to 8600, taking off from where we handled this before. If not, programming continues back to the 10000 traffic director.

## FINISHING OFF THE GAME

It's all over now but the shouting. Everyone has gotten cards, the betting is finished—it's time for the showdown.

*FIVE-CARD DRAW*

```
27000 REM  DECLARATION
27010 GOSUB 47000
27020 FOR X = 1 TO 5
27030 IF P(X) < > 0 THEN 27080
27040 ON X GOSUB 7000,7040,7080,7120,7160
27050 IF X = 5 THEN GOSUB 27100
    : PRINT "YOU'VE GOT " + H$(5) + G$
    : GOTO 27080
27060 IF H(X) > 5 THEN PRINT H$(X) + G$
27070 IF H(X) < = 5 THEN PRINT B$(11) + G$
27080 NEXT
27090 RETURN
```

There's no need for speed now, which is why this segment is way up there in the programming lines. GOSUB 47000 refers to the formatting lines where the backs of the computer player cards are "turned over" and their true values or CD$(X)s are displayed. We do it sequentially for each player still in the game, and then on the chat lines we print the H$(X) of each hand. If they've got less than jacks (27070), we print "I'VE GOT ZILCH"—B$(11). Otherwise we print the straight H$(X) (27060). For the human player, we're still at a loss, because at this point the computer doesn't know the final value of his hand. We wait until now to find it out on the off chance that the human might have dropped out. Why waste time calculating a nonactive player?

*FIVE-CARD DRAW*

```
27100 REM  TRANLATE PREVIOUSLY UNTRANSLATE
        D HANDS IN H$(X)+CHECK(X)
27110 REDO = 1
    : GOSUB 130
    : GOSUB 1360
    : REDO = 0
27120 IF H(X) < 6 THEN H$(5) = "DOODLY SQU
        AT"
    : RETURN
27130 IF H(X) = 6 THEN CHECK(X) = V(X,
        VAL ( MID$ (H$(X),2,1)))
    : H$(X) = "PAIR OF " + V$(X, VAL (
        MID$ (H$(X),2,1))) + "S"
27140 IF H(X) = 7 AND VAL ( MID$ (H$(X),3,
        1)) > VAL ( MID$ (H$(X),5,1))
        THEN CHECK(X) = V(X, VAL ( MID$ (H
        $(X),3,1)))
    : H$(X) = "TWO PAIRS "
27150 IF H(X) = 7 AND VAL ( MID$ (H$(X),3,
        1)) < VAL ( MID$ (H$(X),5,1))
        THEN CHECK(X) = V(X, VAL ( MID$ (H
        $(X),5,1)))
    : H$(X) = "TWO PAIRS"
27160 IF H(X) = 8 THEN CHECK(X) = V(X,
        VAL ( RIGHT$ (H$(X),1)))
    : H$(X) = "THREE " + V$(X, VAL (
        RIGHT$ (H$(X),1))) + "S"
27180 IF H(X) = 10 THEN GOSUB 2740
27190 IF H(X) = 11 OR H(X) = 12 THEN
        GOSUB 32000
27200 RETURN
```

Here's the starting point for the human REDO. It's much the same as before, with the addition of GOSUB 1360 in line 27110 to get an H(5) for the hand. Then we translate the human holdings into a readable H$(5), much as we did before starting at 2820 (one or two slight differences in the routines account for rewriting them rather than using the same one over again). This done, we now know the H(X) for all the hands as well as the CHECK(X)s, which we will now analyze.

```
28000 REM  FIGURE WHO WON
28010 FOR X = 1 TO 5
28020 IF P(X) < > 0 THEN 28070
28030 FOR I = 1 TO 5
28040 IF H(X) < H(I) THEN I = 5
    : NEXT I,X
28050 NEXT
28060 WINNER = X
    : X = 5
28070 NEXT
```

These lines of programming determine the variable WINNER, or WI. We simply run a loop to find out if any H(I) is greater than our H(X) in 28040. If not, we don't necessarily have our winner, but we may. At this point we could have more than one player with the same H(X). All we've determined in 28040 is that no hand values *exceed* this highest H(X), but we've still got to find out if any are equal to it.



```
28080 Y = 2
    : TIE(1) = WINNER
28090 FOR N = 1 TO 5
28100 IF P(N) < > 0 OR N = WINNER THEN 2812
        0
28110 IF H(N) = H(WI) THEN TIE(Y) = N
    : Y = Y + 1
28120 NEXT
28130 IF Y = 2 THEN 28250
```

These lines look for duplicates of the winning high hand, which we now know as H(WI), as in H(WINNER). If we do have duplicates, we know that we will have to check the CHECKs. Here we're looking for how many and which duplicates we have. We set a variable Y as 2, and we initialize the new TIE(1) as WINNER in 28080. We will use Y to count the winners, and we will use TIE(X) to tell us which players they are. In 28110 we look for H(N)s equal to H(WI). If we get any, we increment Y for each one. At the end of the loop, if Y still equals 2, we skip down to the winner's circle—we have only one winner. If Y has grown greater than 2, we know we now have Y-1 players with the same H(X) hand value.



```
28140 FOR X = 1 TO Y - 1
28150 FOR I = 1 TO Y - 1
28160 IF CHECK(TIE(X)) < CHECK(TIE(I))
        THEN I = Y - 1
    : NEXT I,X
28170 NEXT
28180 WINNER = TIE(X)
    : X = 5
    : NEXT
```

```
28190 MULTIWINNER = 2
    : MU(1) = WINNER
28200 FOR N = 1 TO Y - 1
28210 IF TIE(N) = WINNER THEN 28230
28220 IF CH(TIE(N)) = CH(WI) THEN MU(MU) =
        TIE(N)
    : MU = MU + 1
28230 NEXT
28240 IF MU > 2 THEN 28320
```

This may seem complicated, but it's all very logical, and it's the sort of value-sorting problem that could easily come up in your own programming, so try to follow it as closely as you can.

Using a pair of loops, we now check to see who has the highest CHECK value among the winning hands. Again, there may seem to be too many NEXTs, but if you follow the logic by plugging in some numbers you'll see that they are all in the right place in the right number. These loops give us the value of the winning CHECK, but as before, they still don't exclude the possibility of duplicate winners, this time not only with the same H(X) value but also with the same CHECK(X) value! We test for this in 28190–28230. We initialize yet another variable, MU or MULTIWINNER, in addition to the array MU(X) (it's okay to have both MU and MU(X)—the computer will keep them separate, just as it keeps separate H(X) and H$(X) ). Then we run a loop to see if there are any duplicate CHECKs. If so, MU will be incremented, and if MU is greater than 2 in line 28240, more programming will have to be done to solve the problem, but programming of a much easier nature, I assure you.

## *FIVE-CARD DRAW*

```
28250 REM  WINNER'S CIRCLE
28260 ON WINNER GOSUB 46100,46200,46300,464
        00,46500
    : INVERSE
    : FLASH
    : PRINT P$(WINNER) + G$ + G$ + G$
    : NORMAL
28270 GOSUB 63000
    : R(WINNER) = R(WI) + PT
    : PT = 0
28300 GOSUB 7160
    : PRINT "PRESS RETURN TO CONTINUE>";
    : GET C$
    : RETURN
```

First let's look at one winner, plain and simple. Here we know that the winner is player number WINNER, hence 28260. In these statements we are FLASHing the name of the winning player above his cards. Then we give him or her the money in the pot, RETURN, and leave the subroutine. That's when things work out normally. If we have duplicate winners, which is now possible, we do this:

```
28320 IF TPC = 0 AND H(MU(WI)) = 7 THEN 283
        70
28325 FOR N = 1 TO MU - 1
28330 ON MU(N) GOSUB 46100,46200,46300,4640
        0,46500
    : INVERSE
    : FLASH
    : PRINT P$(MU(N)) + G$ + G$ + G$
    : NORMAL
28340 R(MU(N)) = R(MU(N)) + INT (PT / MU -
        1)
28350 NEXT
28360 GOSUB 63000
    : PT = 0
    : GOSUB 7160
    : PRINT "MULTIPLE WINNERS SPLIT POT.PRE
        SS RETURN>";
    : GET C$
    : RETURN
28370 FOR Z = 1 TO 2
28380 FOR I = 1 TO 4
28390 FOR II = 2 TO 5
28400 IF I = II OR V(MU(Z),I) = CH(MU(2)) THEN
        28420
28410 IF V(MU(Z),I) = V(MU(Z),II) THEN CH(M
        U(Z)) = V(MU(Z),I)
28420 NEXT II,I,Z
28430 TI(1) = MU(1)
    : TI(2) = MU(2)
    : Y = 3
    : TPC = 1
    : GOTO 28140
```

In our game we're going to invoke a house rule of sorts that will rarely if ever arise. We are going to define multiple winners as what we want multiple winners to be. If both players hold a pair of aces, they will split the pot regardless of the other cards in their hands. The same is true for two players holding flushes with the same high card: They will split the pot regardless of their other holdings. However, from 28370–28430 we run a further check for two-pair hands, seeing whose second pair is higher if the two highest pairs are equal (the variable TPC is a flag for Two-Pair Check). If we do have multiple winners, we run a flash of all their names, and they split the pot (throwing away any loose change—see the INT in 28340). Now we're ready to clear the decks and deal another hand.

*FIVE-CARD DRAW*

```
29000 HOME
29010 FOR X = 1 TO 5
29020 PRINT
29030 IF P(X) = 10 THEN P(X) = 0
```

```
29040 IF P(X) < > 0 THEN PRINT P$(X) + " IS
        OUT!"
    : GOTO 29090
29050 IF R(X) < 1 AND X = 5 THEN NEXT
    : GOTO 29320
29060 IF R(X) < 5 AND X < > 5 THEN
        PRINT P$(X) + " HAS ONLY $" +
        STR$ (R(X))
    : PRINT "GOODBYE, " + P$(X)
    : P(X) = 1
    : GOTO 29090
29070 IF X < 5 THEN PRINT P$(X) + " HAS $" +
        STR$ (R(X))
29080 IF X = 5 THEN PRINT "YOU HAVE $";R(5)

29090 NEXT
29095 A = FRE (0)
29110 PRINT
    : PRINT
    : INPUT "ANOTHER HAND (Y/N) ?";C$
29130 IF C$ = "N" THEN PRINT
    : PRINT "SEE YOU LATER!"
    : END
29140 IF C$ < > "Y" THEN GOTO 29110
```

First we go HOME. Then we run through each player, displaying his or her bankroll. Then we give the human the option of playing or dropping out. The only new thing here is 29095, the FRE(0) command. FRE(X) is the sort of sophisticated command that gives you arcane information about memory storage that you may never seem to want to know; in fact, it tells you how much memory space is still available to your program. It took me ages just to figure out what it was that FRE actually did, and quite frankly I've never used it this way. But there is a very useful function for this command when expressed as we've done it here. Setting some imaginary variable equal to FRE(0) causes the computer to go through the memory, sweeping away all the cobwebs that may eventually hold us up. You see, every time we put in a new value for a string variable, the old value remains in memory (and let's face it, there are a lot of string variables in "Five-Card Draw"). Sooner or later all the available memory will fill up, and processing will stop while the computer on its own sweeps away the unnecessary values. This could happen at any time, without any explanation—a dead halt, for no apparent reason. So using this FRE(0) equation here, we order the computer to do this housekeeping now, where it won't bother anybody (the player will simply see it as a normal, if irregular, pause). We didn't have to do this in "Space Derelict!" because we used the CLEAR command, which swept away *all* the variables. We can't use CLEAR here because we'll want to save some variables. You don't have to get carried away with FRE—once on the cusp of any program should be enough—but it's nice to put it in to keep control of your computer in your hands rather than having the damned thing seemingly think with a mind of its own.

```
29150 REM  CLEAR THE DECKS
29160 GOSUB 29170
    : GOTO 29290
29170 FOR I = 1 TO 5
    : B(I) = 0
    : LB(I) = 0
    : IN(I) = 0
    : NEXT
29190 BE = 0
29200 CH$ = ""
29210 FOR I = 1 TO 5
    : CH(I) = 0
    : DI(I) = 0
    : BF(I) = 0
    : BL(I) = 0
    : DO$(I) = ""
    : NEXT
29220 FOR I = 1 TO 5
    : F(I) = 0
    : H$(I) = ""
    : H(I) = 0
    : NEXT
29230 NB = 0
    : TPC = 0
29240 FOR I = 0 TO 3
    : NC(I) = 0
    : NEXT
29250 RE = 0
    : SO = 0
29260 FOR I = 1 TO 5
    : TI(I) = 0
    : NEXT
29270 WI = 0
    : ST = 0
29280 RETURN
```

Here we do some more precise deck-clearing, emptying out all the variables we want to empty out. As I said previously, we can't simply use CLEAR, because we need to keep our R(X)s and Dealer (and, in unopened hands, our PT), so we have to go through all of this just to hold one small handful of values. Needless to say, to clear the decks like this, one must know the names of all the variables in the program,* which means a bit more tedium on your part. But it's easy enough to do, and it is extremely important if you don't want your game to go screwy. This section is set up as a subroutine because it is used both here and from earlier in the program where there were no opening hands.

*The DOUBLE-TAKE utility mentioned in the Acknowledgments will print a dump of all the variables and the lines in which they occur—a nice time-saver.

*FIVE-CARD DRAW*

```
29290 D = D + 1
    : IF D = 6 THEN D = 1
29300 IF P(D) < > 0 THEN 29290
29310 POP
    : GOTO 10020
29320 PRINT "SORRY, BUT YOU'RE OUT OF MONEY
      . THAT'S  THE END OF THE GAME."
29330 END
```

Finally, we increment good old D and POP back to 10020 if we have the chips. If not, we'll have come to 29320 from 29050, and we'll learn THAT'S THE END OF THE GAME. And believe it or not, it is. The only thing missing is the high-up pauses:

*FIVE-CARD DRAW*

```
63000 FOR PAUSE = 1 TO 1500
    : NEXT
63010 RETURN
63020 FOR PAUSE = 1 TO 500
    : NEXT
63030 RETURN
```

You might now want to take a week's vacation before proceeding with the next, and final, chapter of the book. I know I'm going to.

# 12

# DEBUGGING THE TOUGH ONES AND OTHER CHOICE TOPICS

## DEBUGGING

Debugging is not much of an issue with short, uncomplicated programs. For that matter, debugging is not much of an issue with long, uncomplicated programs. Look at "Space Derelict!": There all we had to do was run the program, plugging in our play inputs until everything looked the way we wanted it to look. But in effect, with the exception of a few small side roads, our adventure module consists primarily of only one major routine, the conversation with the computer. All the side road sections are small enough and elementary enough to lend themselves to almost instantaneous debugging—either they run or they don't. With the conversation segment, we have the further virtue of using only a handful of variables, with S$, S2$, and C$ taking up the lion's share of them. And what's best of all, these variables are entirely dependent on the player inputs—the C$ that the player plugs in—so to debug, all we have to do is type in every combination of C$ input we can think of and then make the appropriate corrections when an input doesn't work. Although some of those corrections can be complicated, the process itself is relatively straightforward. You might spend a bit of time debugging an adventure, but you'll seldom find yourself stumped on a logic poser. Strategy games, however, will not be so easy to straighten out, so be prepared for interminable debugging sessions easily as long as the original creation of the program.

The main reason for the difficulty in debugging this sort of program is the intrinsic complexity of the program itself. Even if many of the subroutines are not particularly perplexing in their design, they are nonetheless long and unruly. And some of the subroutines, especially the hand-figuring and the subsequent draw and refiguring in "Five-Card Draw," are extremely murky.

There's so much going on in these segments that there's just no way of debugging by plodding through it, as we did with "Space Derelict!": just running it until we get an error, correcting the error, and running it some more. What's worse, we don't have that great time-cutting option of the saved game that we had in our adventure. And worst of all, whereas in our adventure we controlled most of our variables from the keyboard via player inputs, in our strategy game most of the variables are generated by the program itself. The CD$(X)s, the H(X) and H$(X)s, the betting variables—with the exception of a handful of pieces relevant to the human's own hand that are input by the player, all of these are controlled by the program. And to debug successfully, we have to make sure that all of them are controlled accurately.

In addition to getting our program to play correctly, we will also be thinking about flow speed in our debugging stage. When laying down a program at the beginning, it's best to lay it out from line 10 to line whatever without worrying about what subroutine falls where in terms of internal access speed. Then, when you have it all on disk, Renumber it, starting at 10000 with line increments of 10. Now you can shift the slow-moving segments down to the lower reaches below 10000 and keep the ones where time doesn't matter on the high-numbered lines. This is how I worked with the poker program. First I wrote it from start to finish, then I Renumbered starting at 10000, adding that line 10 (GOTO 10000) as my first statement. Then, as I debugged, I made my decisions about which subroutines needed prime real estate and which ones didn't, and I adjusted accordingly using Renumber. It is not impossible to preplan the placements of the fast and the slow segments, but it isn't really necessary: That's why I swear by Renumber—it does a lot of this work for us after the fact, when we can clearly analyze the speed of a routine in action, rather than making us worry about the fact beforehand.

As I said, the worst thing about strategy games is the program-generated variables. You'll recall when we discussed structured programming way back when that one of the reasons for using this method was to aid in debugging. Now you'll see why, when we use our modular subroutines as independent units, debugging one at a time, substituting our own keyboard-generated variables for the program-generated ones. We're going to take one subroutine at a time, debug it, then go on to the next subroutine. And we'll start at the beginning, with line 10000. Our debugging style for "Five-Card Draw," moreover, will be appropriate for any other strategy game.

Subroutines 60000 and 50000 don't require much sweat, because neither of them really does anything all that earthshattering. Here we can simply RUN and see what happens. Since there are no conditionals in these lines, every line will be processed, and syntax errors will immediately pop out at us for correction. That done, we get to line 6000, the shuffle and ante. Here we're going to be looking for two things: first, that we get 40 different cards, and second, that our R(X) bankroll values are decremented correctly. So we're going to plug in a new statement such as this:

```
10025 GOTO 11000
```

which will take us out of our normal flow altogether after the shuffle and ante, and then we'll add some new lines such as this:

```
11000 FOR X = 1 TO 39 STEP 2: PRINT CD$(X), CD$(X+1): NEXT
```

```
11010 GET C$
11020 FOR X = 1 TO 5: PRINT R(X): NEXT
11030 END
```

The first thing this new section of programming will do—and note that this is not part of the actual program but only a temporary debugging tactic—is print out our 40 CD$ values. Line 11000 prints two CD$s on a line; the comma between the CD$(X) and the CD$(X+1) is a screen-formatting command that causes data on the right side of the comma to print on the same vertical line as the data on the left side, but beginning at the middle of the screen. We do this here to get all our 40 variables on the screen at once. We can look at them now and make sure there are no duplicates. If there are, back to the drawing board. If there aren't, hit any key to see what the R(X)s are. They should be 99 (the original 100−1); if so, great; if not, correct. And on from there.

We're going to follow this practice of creating short debugging routines in one way or another for just about every section of the program. In some cases we'll go even further: Not only will we analyze data after the fact, but we'll actually plug in the data first. You can see from this just what we're doing. We're not going to look at the program as a whole. We're just taking it one step at a time, making sure each step works, and then going on to the next step. If we can get all the pieces to work individually, then the whole will take care of itself. After we're finished with each piece we'll delete our debugging programming and save the whole with the latest finished section up and running until we have all the sections ready to go.

Our screen formatting for the deal, to which we branch from 10030, does not have to be fixed in this fashion with a debugging routine. When we get to this subroutine either the blank cards and so forth will end up where they belong or they won't—the results on the screen will be obvious. So we can just run this, see how they fall, and act accordingly. Most likely, any screen formatting, no matter how well planned, won't be perfect the first time you run it, but you'll most likely find one or two elementary mathematical errors that will be easy to correct, and then you'll have what you want.

The next subroutine is the hand-figuring, and we're going to take this segment entirely on its own. Here we have to input our own card values to make sure we get correct H(X) and H$(X)s at the other end. Moreover, we want to do this in a controlled way so we'll be able to check every possibility. We're going to bypass the rest of the program and begin like this:

```
10 DIM C(25), C$(25), S(25), S$(25), CD$(25): GOTO 11000
11000 FOR N = 1 TO 25
11010 PRINT "C(N) − C$(N) − S(N) − S$(N)"
11020 PRINT N: INPUT C(N): INPUT C$(N): INPUT S(N): INPUT S$(N)
11030 NEXT
11040 GOTO 10040
```

This will allow us to type in each individual C, C$, S, and S$ ourselves, which means we can plan them in advance. We can start with each hand possibility and plug in the right cards five at a time for all five hands and then send these values into the hand-figuring routine. The best way to make use of this is to go along with the way this figuring subroutine at line 90 is actually designed: First we might try plugging in various pairs, then we might try different two-pair combinations, threes of a kind, and so forth, all

the way through the insidious three flushes and inside straights. Here's how we catch it at the other end:

```
10045 GOTO 12000
12000 FOR X = 1 TO 5: PRINT H(X),H$(X): NEXT
12010 END
```

Here we see the resulting H(X) and H$(X)s. Since we know what we plugged in, we can compare the hands to the end results. If we get the right H's, we're okay; if we get the wrong ones, we have to figure out why. The important thing is that we are entirely in command of the process, which means that we get (or at least ought to get) exactly what we want. But we have to be very careful to input *every* possible combination. You might put in a hand like this

```
QUEEN SPADE
QUEEN HEART
FOUR CLUB
TWO DIAMOND
NINE SPADE
```

and get the result P12J, which is correct. But this does not necessarily mean *all* pairs will be so collected. At this point in debugging, for all we know the P12 might work and the P35 might not. You have to plug in *every* possibility (or at least thoroughly representative samples thereof) until you're absolutely certain they all work. For instance, I mentioned the DOUBLECHECK$(X) problem earlier, where the way I had figured four straights I could be missing the boat if the first two cards in the hand happened to fall the wrong way. I didn't think of this originally and didn't discover it until I had plugged quite a few four straights into this debug programming and accidentally had come upon this hidden problem. With some four straights, my H's just weren't reading right. A little agony, and DO$(X) came into existence. And by the way, you can see now why the various straight hands got the H$(X) names they did: By putting the correct numbers of each card in order in the H$(X), I was able to make sure not only that I had the straight I was looking for but also that the computer and I agreed about which cards were which that were making up that straight.

The first round of betting, from 10050, was debugged from the program—i.e., I watched it run on its own, with me following the progress of the info box in the corner, keeping notes, making sure that everything went as it was supposed to. Then I'd break into the program to make sure the R(X)s were still where they were supposed to be after the betting was concluded (when you break into a running program, the variable values remain intact—until you make changes in any of the programming lines). Needless to say, when you're debugging you'll need a nice pad of paper in front of you to keep track of what you're doing.

To debug the draw segment I let the computer do the first part: Why should I bother inputting all those card values after I knew the program was debugged up to that point? I simply added some new lines like this:

```
10065 HOME: FOR N = 1 TO 5: PRINT H$(X): NEXT
10066 GET C$
```

and then let the subroutine do its stuff, followed by a new debug program at 11000 like this:

```
10075 GOTO 11000
```

```
11000 FOR X = 26 TO 40: PRINT CD$(X): NEXT
11010 FOR X = 1 TO 5: PRINT H$(X): NEXT
```

This would give me the H$(X)s going in, which I would write down, and then I would get a list of the discards available in order at 11000, followed by a tally of the H$(X)s going out of the draw routine. Knowing what each player had initially, I could tell now whether the discards had been correctly drawn and whether the hands had been amended accordingly. As for the rest of the game, it was pretty much debugged from the actual program, with an occasional break-in to read specific values to make sure everything was on the up and up.

I've simplified a bit here, because every now and then I'd design a debug program a lot more complicated than these, giving me all sorts of information about the hands all at the same time, or else I'd go a lot more deeply into one segment of a given subroutine, hellbent on a very specific task requiring very specific debugging. But the section above is meant as a guide, not a bible, and you should be able to get from it the gist of what we're talking about here. When you're debugging a complicated program, you have to take that program apart and examine every nut and bolt, and this is the best way to do it. Sometimes you'll have to go even deeper, but occasionally you'll only have to scratch the surface. But be prepared to spend a lot of time at this stage of the program. Everything we said in Part One of this book about debugging is doubly true here, and there are no shortcuts in this part of the forest.

## SOUND AND LIGHT

I am no expert on Apple graphics, although I've gone so far as to design my own fairly elaborate shape table generating program. As I've said before, graphics must remain beyond the scope of this book, primarily because the subject is too complicated to get the justice it deserves in the small amount of space we would be able to give it. But whereas a good case can be made against using graphics in adventure games, it's a little harder to say that TWO DIAMOND on the screen looks better than a drawn 2 and a cute little diamond placed neatly on a rectangular card. If you are serious about graphics, your best bet is to purchase a good graphics utility. A solid graphics package ought to give you not only everything you need to create art as complicated as you want it but also should allow you to do so without having to know all the nuts and bolts of exactly what you are doing. Because, take my word for it, graphics are a tough business, and as the state of the art improves, it is one that is less and less accessible to the Sunday hacker—unless that hacker arms himself with a state-of-the-art utility. Let someone else worry about how a color-fill routine works; all you want to do is use it toward an end, unless you're interested enough to devote the time and energy necessary to conquer a very difficult machine-level study.

The major problem with graphics for the Basic programmer is that a knowledge of Basic is not enough. Programming in Basic seldom requires any knowledge of the memory map of the Apple, for instance, which is your first concern with any halfway decent graphics routine. And you can program

in Basic until kingdom come and never have to translate shapes into vectors, then vectors into bits, and then turn the resulting bytes of bits backward into hex, which is how you save a shape table. Nor should you have to know all this. But if you want graphics, then you have to change your tune and, at least minimally, get some feel for this complicated folderol. That's what we're going to talk about in this section.

Not far removed from graphics, and a good starting point for this discussion, is the subject of sound effects. In "Five-Card Draw" we do use sound, but only the ringing of G$. What if you want more or different sounds? There's a way of getting them, and it's not very complicated, but it requires a machine-language program, or at least a machine-language program masquerading as a Basic program. Here it is:

```
10    HOME
20    FOR Z = 770 TO 790
    : READ D
    : POKE Z,D
    : NEXT
    : DATA   173,48,192,136,208,5,206,1,3,2
        40,9,202 ,208,245,174,0,3,76,2,3,96

30    INPUT "FREQUENCY?";F
    : POKE 768,F
40    INPUT "DURATION?";D
    : POKE 769,D
50    CALL 770
60    HOME
    : GOTO 30
```

Yikes! Talk about a meaningless batch of numbers. But don't blanch at these apparent Egyptian hieroglyphics yet: They do a lot, all of it good.

The real sum of the program is line 20. Here we're writing a short machine-language program and POKEing it into memory. We start at address 770 (decimal) through 790, using the DATA values provided at the end of line 20. This machine-language routine takes the values of addresses 768 and 769, which we will POKE in in lines 30 and 40, and turns them into sound. After each sound is made, processing will return to the Basic program from which this routine was CALLed. Here our CALL is in line 50 (remember, a CALL sends processing out of a Basic program to a machine address specified by the number following the CALL command). Processing then goes on to line 60 after each sound is made. Before we continue our discussion, it would be best if you entered this short program on your machine; what we're going to say next will make much more sense if you can hear this program in action. . . .

All set? All right, RUN the program. The first thing you see is the word FREQUENCY? at the top of the screen. Type in any number from 1 to 255, then Return. Now you'll see the word DURATION? under FREQUENCY? Again, type in any number from 1 to 255. When you press Return you will hear a tone, then the screen will clear, and once again you'll get the INPUT

requests. Experiment for a while. Any and all the sound effects you've ever heard in any Apple program were generated with a routine essentially the same as this one.

If you've read your manuals carefully, you know that there is a speaker inside the Apple that clicks when you toggle the right switch (address −16336). The machine routine we've just written allows us to toggle that switch to get something more than just clicks. The frequency, our value we POKE into 768, boils down to whether the tone is high (a low-frequency number) or low (a high-frequency number). The duration we POKE into 769 is just that, the length of time we hear the sound. Manipulating these two values gives us the range to create everything from music to gunshots. In our sample program we get only one tone at a time, and then we have to INPUT values for the next tone. But what if we created some sort of loop to make extended sounds for us? Try deleting lines 30–60 and inserting something like this:

```
D = 30
POKE 769, D
FOR F = 1 TO 255
POKE 768,F
CALL 770
NEXT
```

Here you'll get a bona fide sound effect, not unlike a UFO landing on top of your keyboard. Not bad, eh? Change it to:

```
FOR F = 255 TO 1 STEP −1
```

and you'll get to hear the UFO taking off. Using loops incorporating either F or D or both will afford you endless possibilities. I've come up with quite a few myself now and then, and it's a lot of fun. And as you learn the relationships of F to D and how the different values make different noises, you'll eventually be able to re-create any sound you want that the Apple is capable of. Just remember that you can only POKE in integer values between 0 and 255. The span 0–255 represents the size of any one byte (00000000–1111111, or in hex, $0–$FF); bigger or smaller than that, you're machine can't chew. Once you've POKEd in the machine routine, you can access it with a CALL from any point within any other Basic program, to manipulate any D's and F's you can possibly arrange. The machine routine will stay put until you reset the computer.

I won't go farther into the workings of sound because you have to hear it to believe it, and you can have a lot of fun playing with it yourself. But I do want to point out one last thing about this machine-language routine, and that is its placement in memory. The lowest line we POKE is 768 (in line 30), going up to 790 (the loop in line 20). Not very much at all, and as it turns out here just happens to be a little bit of free memory real estate starting at $300 (or decimal 768) that is used frequently for this sort of short machine-language routine. This real estate is always available to us (provided you haven't already loaded another machine-language routine there), but unfortunately there's not enough space to hold, say, a whole full-blown machine-language program, or even, more to the point, a decent-sized shape table. And that problem of storing a shape table is important when we get into graphics.

So now you can make sounds, and with experimentation you ought to be able to reproduce the shuffling and dealing of cards. But you say you want to *see* a hand of cards? All right, here it is.

```
10    DIM C(52),S(52)
20    REM  CREATE SHAPE TABLE
30    FOR X = 7569 TO 7964
40    READ D
    : POKE X,D
50    NEXT
60    POKE 232,145
    : POKE 233,29
70    HGR
80    HCOLOR= 3
90    SCALE= 1
    : ROT= 1
100   REM  SHUFFLE
110   C(1) = INT ( RND (1) * 52) + 1
120   FOR X = 2 TO 5
130   C(X) = INT ( RND (1) * 52) + 1
140   FOR Y = 1 TO X - 1
150   IF C(Y) = C(X) THEN Y = X - 1
    : NEXT
    : GOTO 130
160   NEXT Y
170   NEXT
180   FOR X = 1 TO 5
190   S(X) = INT ((C(X) - 1) / 13) + 1
200   IF C(X) > 13 THEN C(X) = C(X) - 13
    : GOTO 200
210   IF C(X) = 1 THEN C(X) = 14
220   NEXT
230   N = 1
    : M = 1
240   FOR X = 1 TO 5
250   IF X < 5 THEN DRAW 19 AT N,M
260   IF X = 5 THEN DRAW 1 AT N,M
270   DRAW C(X) AT N + 2,M + 2
280   DRAW S(X) + 14 AT N + 9,M + 2
290   N = N + 10
    : M = M + 10
300   NEXT
310   FOR PAUSE = 1 TO 5000
    : NEXT
320   TEXT
    : HGR
    : GOTO 100
330   REM  SHAPE TABLE PARAMETERS
340   DATA    19,0,40,0,90,0,  102  ,0,  11
        1  ,0,  126  ,0,137,0,151,0,159,0,1
        75,0,191,0,199,0,208,0,222,0,239,0
350   DATA             0,1,23,1,41,1,62,1,8
        3,1
360   REM  FULL CARD OUTLINE
```

```
370   DATA  45,45,45,45,45,45,45,45,45,53,5
        4,54,54,54,54,54,54,54,54,54,54,54,
        54,54,63,63,63,63,63,63,63,63,63,39
        ,36,36,36,36,36,36,36,36,36,36,36,3
        6,36,36,4,0
380   REM  2 THROUGH ACE
390   DATA  18,12,12,117,246,30,30,30,45,45
        ,5,0
400   DATA  10,12,117,246,14,246,231,28,0
410   DATA  73,54,54,54,6,128,128,8,63,63,1
        00,12,12,12,0
420   DATA   73,57,63,55,54,45,117,246,63,6
        3,0
430   DATA  73,10,28,63,30,54,46,12,117,246
        ,63,28,4,0
440   DATA  45,45,246,30,30,30,54,0
450   DATA  41,117,30,30,14,14,30,63,28,12,
        12,28,28,12,5,0
460   DATA  41,117,246,30,30,30,30,8,8,8,8,
        63,28,100,5,0
470   DATA  45,45,221,51,54,54,54,0
480   DATA  73,49,54,54,30,63,28,36,0
490   DATA  41,117,54,246,53,14,28,63,231,3
        6,36,12,5,0
500   DATA  54,54,54,6,128,128,96,12,12,12,
        140,211,154,114,14,14,0
510   DATA   9,30,30,54,54,6,72,9,36,63,127
        ,73,36,28,28,28,0
520   REM  CLUB,DIAMOND,HEART,SPADE
530   DATA  73,241,45,62,119,14,12,12,46,62
        ,230,63,231,28,30,245,53,12,45,246,
        45,5,0
540   DATA  73,241,45,14,63,63,30,45,45,45,
        30,63,63,14,45,30,30,0
550   DATA        9,14,63,46,45,12,12,14,247
        ,45,62,63,63,119,45,45,30,63,14,14,
        0
560   DATA  73,241,45,14,63,63,30,45,45,45,
        14,63,63,63,63,45,14,45,62,63,0
570   REM  SMALL OUTLINE
580   DATA  54,54,54,54,54,54,54,54,54,54,5
        4,54,54,54,46,45,45,45,45,5,128,128
        ,128,128,128,128,128,128,128,128,12
        8,128,128,128,128,128,128,128,72,73
        ,73,36,36,36,36,36,63,63,63,63,63,6
        3,63,63,63,7,0
```

That, my friend, is a hand of cards. Any wonder I didn't want to go too heavily into graphics?

In this long and tedious program we're creating a shape table and storing it in memory, then pulling out the appropriate shapes for each hand dealt. In case you don't want to enter this program (and I don't blame you), I'll tell you that the end result on the screen is simply five rectangular cards laid one on top of the other, the values of the cards in numbers or letters (2, 3, 4, . . . T, J, Q, K, A) and a pretty little suit symbol. All five hands in our

poker program could have been laid out in this manner if we had been so inclined.

Lines 30–50 POKE in the DATA values (395 of them) into addresses 7569–7964. These addresses were chosen because, in this small program, they are in the middle of nowhere. You have to be careful POKEing in this many addresses because your Applesoft program or variables can run into them and you can have your program cannibalize itself if you're careless about it. When locating a shape table, make sure it's in a totally secure area. In line 60 we tell the computer where we have put the shape table. Address 7569 decimal is $1D91 in hex; we turn that around to $91 and $1D and then translate $91 into 145 decimal and $1D into 29 decimal, and then POKE those values into 232–233. Since the computer reads numbers backward (the low-order byte, as it is called, $91 in this case, followed by the high-order byte, $1D), we always have to make this confusing (to us) transformation. Then we shuffle the cards in 100–220 as we did before, and then using the DRAW command, we get the appropriate cards onto the screen. DRAW is used only with shape tables, so a word about both is in order.

The concept of a shape table is relatively self-explanatory. It is a collection of numbers that the computer can translate into a series of pictures, from picture 1 on up. Our picture 1 in our shape table happens to be the plain rectangle that will be the border of our card, so when we say DRAW 1 AT N,M in line 260 we are saying put shape 1 at location N,M (our N and M, determined by lines 230 and 290, are positioned within the regular boundaries of the high-resolution screen set in line 70). Conveniently enough, in this table shapes 2 through 14 are the numbers and letters from 2 through A; hence line 270, where we can DRAW this character at such and such a place. Our suit pictures are located in the table from 15 through 18; hence line 280. And the last shape, number 19, is part of a rectangle, so it will look as if the cards are piled one on top of the other in perspective. When we're finished drawing a hand, the entire screen is cleared in 320, the drawing run again starting with a new deal at line 100.

To create a shape table such as this one requires that you first draw your shapes on graph paper, then turn those shapes into a series of numbers. You will need a program to do that translation for you, and you can either design your own by trying to make heads and tails out of the shape table discussion in your manuals (good luck!), or you can buy one, or copy one from a magazine (shape drawers pop up regularly in them). Or, if all you want is a deck of cards, you can copy what I have here. This shape table was generated the easy way, using diagonal plotting, which means that these shapes will not SCALE up or ROTate (you can't double or triple their size or twist them around—they'll just turn into gibberish), and will print only in white.

I designed this program so you could take the parts you like and leave the parts you don't like. After the parameters of the table are established in 340–350 (this tells the computer first how many shapes there are—19—followed by how far away they are from the first byte in the table—40 bytes, 90 bytes, etc.), each shape is given a separate line. If you don't like the way I drew my diamond, for example, you can change line 540 yourself, putting in your own diamond, and so forth. Note that each shape determination ends with a 0: This tells the computer that it has reached the end of the shape, so it knows when to stop DRAWing.

There is one intrinsic flaw to this shape table that has nothing to do with the shape table but with the problem of mixing shapes and regular text. You can't do it. It is impossible to have the computer print its regular characters—the numbers and letters and symbols you normally see on a TEXT screen—onto a high-resolution screen. Once we're in HGR or HGR2, the only things we can see are the things we make ourselves. This means that if we want to print text messages such as "I'LL BET $1", we can't do it unless those words or letters are stored in our shape table (except in those cases where you're willing to limit yourself to the four lines at the bottom of the HGR screen, and even these lines are normally unavailable in HGR2). So if you wanted to redesign our poker program using graphics, you would either have to have all your chat at the bottom of the screen (not desirable), or add to your shape table all the characters you'll be using as full phrase lines of text, or else simply store the entire alphabet and the numbers 0 through 9 at the end of the shape table and then add some sort of character generator to get those characters on the screen where you want them. I did not, as you can see, go so far as to draw the entire alphabet as a shape table, but again, such ready-made programs are easily available in stores and in magazines. Be forewarned: If you type these programs yourself, they are incredibly long and tedious. But you'll either have to pay or slave to get both pictures and text on the high-resolution screen.

## OTHER STRATEGY GAMES

My expressed purpose in Part Two of this book has been to provide you with a complete and fairly complicated strategy game in the hopes that seeing how the problems were solved in it would help you solve problems in any other strategy game you might design yourself. This means that short of designing a different brand of poker, in which case all you would have to do is co-opt this program, changing a few small parts, you're pretty much on your own. But we can still talk about a few ideas for some other games and see how what we've discussed so far can be used directly in designing those programs. And the best place to start is with other card games.

Possibly the easiest game you can design is blackjack, because there's no need to incorporate any strategy for the computer. In casino blackjack the dealer must rigidly adhere to the rule of taking a card on 16 and standing on 17 (although some casinos allow the dealer to hit a "soft" 17—i.e., a hand where an ace is counted as 11, as in ACE CLUB, SIX SPADE). All you have to do is make the computer the dealer and provide a strict accounting of the values of the hands to figure out what everyone is doing. Then you figure out the winner, pay out or take in the money, and deal another hand. In an elementary program, one player vs. the computer/dealer, the only real programming complication would be accounting for aces, which can have a value of either 1 or 11, at the discretion of the player. What you might do is default aces as 11. Then

H(X) = V(X,1) + V(X,2)

to give you the original H(X) of a hand. Then with each hit:

H(X) = H(X) + V(X,N): IF V(X,N) = 11 AND H(X) > 21 THEN H(X) = H(X) − 10

This will give you the highest possible H(X) under 21. Aside from this, all you have to do in evaluating the hands is add up the V(X,N) values, calling a bust when it goes over 21. Player cards would be dealt as the player requests them: whether the dealer takes any additional cards would be determined by the value of the dealer's hand.

By now you may know me well enough to guess that I personally would not be satisfied with a blackjack game where just one player sat down and played against the computer, and the one who got closest to 21 would win. Too simple, and not a real simulation of what casino blackjack is all about. It would be essential that the play allow options for doubling, pair splitting, surrender, and insurance, at the very least (these are all betting strategies, variations of which are usually available at the casino tables). There's no fun to betting unless you're allowed to make all the bets. Second, in real casino play the house uses anywhere from one to eight decks at a time; this too should be an option, perhaps chosen at the beginning of the game. Needless to say, such a long shuffle of many decks would require some hiding to mask slowness, or else a speedier routine, if you've got one. Third, the player should be allowed to play more than one hand at a time, or perhaps the program would allow for more than one player so that up to seven people could play at once against the machine. Again, this is just like real life, and the more computer games of this nature resemble real life, the better they are. These additions to the program would most definitely make it that much sharper and that much more fun.

There is one last facet of blackjack, which I address to students of the game. You may not know it, but there is a large body of literature available on the game of blackjack. The reason for this is that for the diligent player there are a number of systems that will beat the casino odds. The percentage in the player's favor is somewhere around 1 percent, but over the long term any percentage will earn money. Unfortunately, to be able to play and get these odds on your side, you must not only play your cards with exceptional precision, but you also must "count" the values of the cards as they are dealt. Precision play isn't too hard, because all you have to do is memorize a few dozen hand combinations and always play according to those combinations. A man named Edward O. Thorp originally introduced these combinations to the world in a best-selling book called *Beat the Dealer.* These combinations are called Basic Strategy, and they will entirely determine whether you will stand or draw a card, based on the total value of your hand against the up card the dealer is showing. The other side of the coin is the counting of values, and there are numerous sytems for this. Usually this requires keeping track of all the aces and 10-value cards (10 through king) as they are dealt in ratio to all the other cards. This means you will have a running tally in your head something like $+2$ or $-1$ or 0 or $-4$. This number will tell you whether to increase or decrease your bet and by how much. For more on both Basic Strategy and counting, any of those books on the blackjack shelf will get you started in the right direction.

Anyhow, wouldn't it be interesting to program some of this information into your own game? You could have a computer-generated player who always bets Basic Strategy, and you could see for yourself how he does over the long term. You could even have this computer-generated player make his bets on the basis of a counting system. Or you could program a running tally

of the count to be displayed on the screen as a tutorial. The possibilities are fascinating, and they are not outside the realm of the Basic programmer with a real interest in blackjack. By the way, I have seen ads for a blackjack program that does all of this and much more, written by a man named Ken Uston, a well-known blackjack writer and player. If you'd rather have someone else tangle with this programming while you just reap the benefits, you might want to look into what this expert has to offer.

On the other side of the spectrum from blackjack, in both programming and play, is the game of bridge, considered by many to be the ultimate card game. Right at the top, the biggest problem you might have in tackling bridge is the size of the program—you could probably eat up 7 or 8 megabytes if you really got carried away with it. I have seen one bridge program in the public domain that is actually two programs, one to deal and bid, the other to play. The values from the first program are simply stored in a file when that segment of the game is over and then recalled at the beginning of the second segment. Not a bad idea to consider yourself. I've never looked inside this Basic program, but I have some of my own ideas on how I would do it if I were to attack such a monster.

The bidding side of it wouldn't be too hard, regardless of the bidding system used. Bridge bidding is fairly rigid; in the good old Standard American I used to play in college, for instance, you would only bid "one no trump" with an evenly distributed hand, 16–18 points. If you had even distribution and 20 points, one no trump was out of the question. And so forth. Every hand, any bidding system, has specific predetermined right and wrong bids; in your bidding segment you would define your hands both by point count, an H(X) value perhaps, and distribution, perhaps a D(X,Y), where X is the player and Y is the suit from 1 to 4, and the value of D(X,Y) would be the number of cards for that player in that suit. Not bad, and it would answer just about everything you need to know. All you would have to do is program an algorithm where, given all the possible H(X) and D(X,Y) values, you would instantly come up with the appropriate bid. Remember, the computer would have to bid for three hands, while the human player would be responsible for only one of them.

The hardest part about designing bridge would be the algorithms of play. While the human played his own hand and the dummy, you would have to handle the two opponents. Here your program's success would rest very much on the two essentials of strategy-game design: your complete knowledge of the best strategy combined with your ability to translate that strategy into computer play. You would want to put in as much as possible: the ability for the opponents to play their leads in response to the bidding, in response to the dummy, and even in response to each other. There are so many possibilities for all of these that you can easily see why, as of this date, the critics say that the ultimate bridge program has yet to be written (which is odd, because they've done some amazing things with chess—maybe there just haven't been any grand master bridge players who have gotten around to buying themselves a computer yet). Maybe you could be the one to do it. Why not?

There are dozens of other games out there waiting to be programmed. And if you're an avid player of any of them, the successful design of a challenging computer version can be an incredible satisfaction. Backgammon? Othello?

Go? Sure, these are all available in store-bought programs, but although they may be better than anything you could ever design yourself (and that's a very big maybe—we're nowhere near the end of the strategy-game-software spectrum at this early date), none of them can provide the thrill of doing it yourself and doing it well. For me, the ultimate challenge is Monopoly—the speed of the computer to make the moves, nice graphics, but most of all, killer algorithms to control the wheeling and dealing segments of the game. It's definitely on my list of projects to come (you can put it on yours, too, if you like; the more the merrier). I get to play the real game only at Christmastime, after using the holidays as an excuse to gather enough players at one time to get a game up. But if I programmed it, I could be playing right now. . . .

## THE ULTIMATE STRATEGY GAME

The ultimate strategy game? Is there really such an animal? In fact, there is: It's the strategy game you invent yourself, a completely original computer game entirely of your own design. It's not a simulation of any other game, or a new version of some other game. It's new, it's exciting, it's yours. And it can be the most satisfying programming you'll ever do.

The idea of inventing your own game is very daunting. It's not the same as creating an adventure, because even though any adventure you design will be original in plot, it will still be an adventure, a new installment in an already-existing genre. But creating a new strategy game means introducing something that never existed before. Once upon a time chess, checkers, Go, Monopoly, bridge, backgammon—you name it—did not exist. Someone had to invent them. Usually that someone relied on concepts that had come before (protochess games seem to go back almost to the beginning of recorded history, for instance, and there were board games on the market long before Monopoly came out), but nonetheless some inventor looked around, and with the ideas of the past to guide him, developed something new and lasting. Am I making this all sound too grandiose and overdramatic? Perhaps, but to invent a new game that's fun to play is no mean feat and should be relegated the respect due such an undertaking.

With the advent of the personal computer came a whole bunch of new, unique games designed to take advantage of the computer's unusual gaming personality. The adventure game and the arcade game, of course, are the most original of these, in that they did not exist before computers were there to play them, and without computers (or at least computer chips) they could not exist at all. And more to the point, they have proven their durability against the criterion of sheer fun. But in the earliest years (the Dark Ages of the mid-to-late seventies), many other new games were sent down the pike as well, if not necessarily all that exciting to play, at least original and unique to the computer. One genre that comes to mind is what I think of as the king and the wheat—the player takes the role of the leader of a country, and against various adversities must make sure that the gross national product of the domain exceeds the national debt (our most recent presidents could learn something from this sort of game). Then there's the genre I call the impenetrable war module, where you control one faction and the computer controls

the rest, and your job is to figure out the rules before you lose the game. Usually these confusing games take place out in the galaxy somewhere and have as many possible moves as there are keys on the computer. All of these games are in the public domain and are not hard to come by, either from other users or from computer store software libraries, which the owners usually make available to their hardware buyers. And for the most part, all these games have one thing in common: They aren't much fun to play. As exercises in programming they might have been satisfying to their creators (something I highly endorse), and as finished programs they may be edifying to other programmers (again, nothing to sneeze at), but as games per se they leave a lot to be desired: Five minutes later you're ready for another one. To be honest, I too have tried to create one of these talky, rule-heavy games, which after a few months I finally abandoned because when a friend of mine tried out the first stage that had already been programmed he fell asleep on the keyboard.

So what is the key to creating a really good new game? The traditional game industry, those people who market new board games and whatnot, has had a bylaw almost as old as the industry itself, and that bylaw has already entered the parlance of computer game people: easy to learn, difficult to master. There are axioms and corollaries to this law, but they all hearken back to that one dynamic: easy to learn, difficult to master. It's the litmus test that any successful game can pass with flying colors. Look at poker, for instance. You get a given number of cards, and you try to match as many as you can, either by suit or by value. That's all there is to it. Checkers—you move your pieces one at a time to the opposite end of the board, jumping the other player's pieces along the way. Ditto Chinese checkers. Backgammon—you try to move your pieces to the opposite end of the board before the other player does (it almost sounds like checkers when you put it that way). Even chess has only six different pieces, each with one or two rigidly defined moves on a static playing field; all you have to do is trap one particular piece of your opponent's before he traps that same piece of yours. Nothing to it, when you boil it down to its simplest elements. All of these games, even chess, can be learned in a few minutes. Most of them can even be played in a few minutes. And all of them are endlessly fascinating and have withstood the test of time. That's the sort of thing you want to accomplish with the game you invented. Easy to learn, difficult to master.

The best way to develop this sort of game is on paper. After you get a few of the basic elements set in your mind, forget about the computer for a while and see if the game will play. Cut out some cardboard pieces and play it, adjusting, changing, fixing as the game develops. Try different pieces, different moves, different playing fields. Add new rules, modify old ones, simplify as much as possible. Get it down to something that really plays and then learn to play it, because your program is going to have to know perfect strategy to provide the player with a good opponent. Only after you've done all this are you ready to lay down your first programming statement.

One thing to avoid at all costs is randomness. Short of something like a deal of the cards, with its almost infinite variations, random factors do nothing but detract from a strategy game. The more random factors there are, the less strategy there can be, by definition.

A nice thing to put into your game (whether your creation or otherwise) is

a set of play levels, from the easiest to the hardest. This gives the human player a chance to start easy and graduate to the more difficult stuff. And it's not that difficult to do. Simply provide an input early on for the player to choose his option and then set a level flag in the program accordingly. Then use that flag to branch to strategic/tactical subroutines in the program, one subroutine for each level. Your algorithms in each subroutine would get progressively more sophisticated as you go along, without necessarily changing their basic design. In a way we did something very much like this in our poker program by creating four players with different styles of play. Most of the program plays the same regardless of which player we're talking about; it is only in those special lines where we assigned the DF(X), RF(X), and BF(X) values, based on the content of the hands, that anything changed all that much. Imagine that we had designed this game where the human played only against one computer opponent at a time. In that case we would have given the human four options, play levels 1 to 4. At level 1 we'd have given him Marvin; level 2, Betsy; level 3, Walter; and level 4, Gerry. Not all so difficult, when you think of it that way. Indeed, in some games level changes might require more than just the tinkering of three variables, but trying to keep level changes to the minimum number of variables would have to be one of your first priorities. But if you needed other changes along the way, your level flags would do the job for you. Let's say that your level flag is the variable LF. Whenever you need to differentiate play on the basis of level you would simply program ON LF GOTO or IF LF > 3 THEN or somesuch. It can and should be done if you want your game to be all it can be.

To give you an idea of an original strategy game that did not exist before someone invented it on the computer, look into the Penguin Software program "Pensate" by John Besnard, which came out while I was in the middle of writing this book. A review in *Softalk* was intriguing, so I bought the program and played it, and yes, indeed, it was exactly the sort of original game that could serve as a model for anyone thinking of rolling his own—easy to play, difficult to master, and fast in the bargain. The premise is simple. You have a playing piece on an 8x8 grid. Your object is to move that piece to the top without hitting or being hit by a computer piece. The computer has up to 10 different pieces, each of which moves differently but all in clearly defined, predetermined directions. You also must enter from two to four moves at one time, which complicates things nicely.

Although this may sound a bit obtuse, it is quite easy when you read the directions in the box, and you can start playing quite successfully from the minute you open the package. But the game gets harder and harder as you continue to play it. And it becomes positively addictive. The thing is, the computer strategy is clearly laid out in front of you. You know how all the pieces move, so all you have to do is predict those moves so your piece can travel accordingly. There's no real secret, so your success at the game is entirely dependent on your own thinking. No random factors, no hand-eye coordination, no mysteries—just you vs. the computer. Fun and intriguing, and the perfect starting point for anyone seriously thinking of creating his own games.

The final thing I should mention about doing it yourself concerns graphics. Again, graphics are difficult to program, but most likely they will be essential in your self-made game. If possible, try to program in low-resolution graph-

ics, which if not so attractive on the screen are at least that much easier to program. With hi-res you'll have all the problems of memory-mapping and variables eating shape tables and vice versa, things not a factor in low-res with its simple PLOT commands. Graphics, high or low, are nice, but they require much drudgery and much programming sophistication. Proceed at your own risk.

## CONCLUSION

"Mother of mercy, is this the end of Rico?" I'm afraid it is, except for the Glossary. We've come a long way in these few hundred pages, and we've covered a lot of territory. We've gone over a couple of megaprograms that eat up just about all the memory we have there for the eating, and we've talked about everything I could think of to help you program similar—perhaps even better—programs yourself. Some of it has been cockamamy, my own personal style, and some of it has been the wisdom of the ages. A lot of it has been somewhere in between. And now I'm at a loss to tie it all up in one neat little package. I'd like to end on some profound note, but nothing even remotely profound seems to come to mind. I guess I've just plain and simple shot my wad.

My final word, in the time-honored tradition of computer people everywhere, is this: If you have any comments, questions, fixes, or changes in the programs you'd like to share, improvements perhaps for future versions, anything at all, let me know. The publisher's address is on the title page of this book, and even though I'm getting out of town on the next stage, they will always know how to reach me and will forward any correspondence. By the time this book is in your hands I'll probably have a few new ideas myself. We might have some fun sharing them.

Cheers!

# GLOSSARY

**adventure game.** A computer game where a player (or players) tries to solve a complex puzzle by traveling through the game universe and successfully manipulating the objects therein. In this book we differentiate between treasure hunts, in which the player tries to amass riches, and situational adventures, where the player has to perform a certain complicated act to win the game.

**algorithm.** A formula or series of formulas or a section of programming that boils down a complicated computing process into one tight quantification of data.

**AND.** A Boolean operator. In an AND statement, both sides of the AND must be true for the whole condition to be considered true. *Compare* OR and NOT.

**array.** The method by which one variable name can be extended to cover numerous situations; arrays are usually used when variables are related—for instance, door 1 through 17 represented as D(1) through D(17). This example is a simple array. You could index more complicated arrays such as V(1,2) or S(3,4), which we do in this book, or even something like ZI(4,2,5,3,10,24), although it is unlikely you would be able to make any sense out of it afterward. Variables can be used instead of numbers within those parentheses; the maximum number of values you can assign to your array is called the dimension (*q.v.*).

**ASCII.** All the characters on the keyboard, including Control characters, Escape, etc., are understood by the computer only as bytes, values from 0 to 255. These values are the ASCII character codes. Applesoft uses about half of them; see your manual appendices for more information.

**assembler.** A program that translates mnemonic commands into hexadecimal machine-language programs.

**AT.** *See* DRAW.
**binary number.** Binary numbers are based on the number 2 and are the native vocabulary of a digital computer. The 6502 processor in the Apple traffics in 8-bit information—i.e., binary numbers 8 characters long.
**bit.** *See* byte.
**BLOAD.** The command to load a binary file (e.g., a machine-language program such as a shape table) into memory.
**Boolean logic.** The correct manipulation of AND and OR operators and all combinations thereof.
**branch.** Branching is the act of going from one part of a program to another, through either a GOTO or a GOSUB.
**break.** Stop a running program. When you break into a program all the variables remain live—i.e., they all hold their most recent values—until you change something, at which point the variables are cleared.
**BSAVE.** The command to save a binary file (e.g., a machine-language program, a shape table) onto a disk.
**bug.** An error of logic that causes your program to run awry. A bug should be considered the next generation away from simple syntax errors or other, similar mistakes in programming that the Basic Interpreter will catch on its own. Bugs are the errors you'll have to catch yourself.
**byte.** A byte consists of 8 bits, and each bit is a binary number, 0 or 1. So a byte would look like this: 00011001. However, when your Apple talks to you it usually speaks decimal, not binary, so a byte becomes any value from 0 to 255 (or from 00000000 to 11111111, the maximum byte). This information plays no part in the plot of this book.
**CALL.** The command (usually from within a Basic program) telling the computer to go to a specific memory address and perform the routine that begins at that address. CALL 770, for instance, would send the processing to whatever bit of machine-language business was located at address 770 and following. As a rule, after CALLs are executed, the machine routine is performed, and then processing returns back to the Basic program.
**carriage return.** The end of a program line or an INPUT, so noted by pressing the Return key.
**CATALOG.** The command that calls forth the directory of programs on a disk.
**CHR$.** CHR$(X) takes the character in the parentheses and returns its ASCII (*q.v.*) value.
**CLEAR.** The command to reset all variables within a program to 0.
**CLOSE.** The command to put away a text file for another day.
**colon (:).** Not a part of human anatomy but an important punctuation in Basic statements. A colon can separate multiple statements on one line. But note that if any statement on the line is a conditional (IF . . . THEN), processing will continue past the colon only if the condition is met.
**command.** Anything you tell the computer to do immediately, also known as direct or immediate command. *Compare* statement.
**compiler.** A program that translates a high-level language program into machine language; it is the compiled program, and not the high-level one, that is eventually run on the computer.
**complication.** In an adventure game, a complication is anything the player has to figure out to proceed. The more complications in an adventure, the better.

**concatenation.** The joining together of two disparate items. Multiple strings are concatenated by + symbols (A$+B$). A string is concatenated to a number by a semicolon (INPUT "HOW MUCH?";X). Variations on these themes abound.
**conditional.** A statement that begins with IF. Whatever is between the IF and the THEN is the condition.
**copy protection.** The process of scrambling data on a disk to keep people from copying the data.
**counter.** A variable in a program that has no particular meaning of its own. A counter variable is used to move through loops where the variable itself is of no importance—e.g., FOR X=1 TO 5: IF IMPORTANTVARIABLE = X THEN Z = 10: NEXT. X is the counter in this statement.
**cowcatcher.** Any programming that picks up and acts on all the conditions already excluded by the previous programming. This could be anything from an error trap (when you prompt for a "Y" or an "N" and the user inputs "W") to a conversational response in an adventure.
**CPU.** Central Processing Unit. The real name for the thing under your Apple keyboard that does the computing.
**cursor.** The blinking white square that tells you where within the text window the computer thinks you are. As a rule, the computer will always be right about this. HTAB and VTAB (*q.v.*) commands can control placement of the cursor.
**DATA.** In a program, DATA lines contain just that, a collection of data to be READ (*q.v.*) by the program when the time comes. DATA lines can be placed anywhere within a program without affecting processing, and string data need not be so denoted by quotation marks.
**debugging.** The process of going through a program and taking out the bugs. This can take anywhere from 50 to 1000 percent of the original time spent programming.
**default.** The normal setting or value for a machine address, variable, or operation. For example, undimensioned arrays default to 11 values—i.e. X(10).
**DELETE.** A DOS command to remove a file from a disk. In fact, all it does is remove the file name from the directory, but that's good enough for most of us.
**DIM.** The command for setting the extent of your arrays. The format is DIM X(27), for example. With complicated arrays the format could be DIM X(4,10), or DIM X(9,9,9), and so forth. Dimensions of arrays default at 11 values (e.g., DIM X(10), since the zero value is counted), so you don't have to dimension arrays of this size or less. One-line statements such as
DIM X(24), Y(19), Z(4,4)
are perfectly acceptable time- and space-savers.
**dimension.** *See* DIM.
**DOS.** Disk Operating System. The traffic cop in your machine memory that controls movement between the disks and the CPU.
**DRAW.** Command to draw a given shape from a shape table. DRAW N AT X,Y would draw figure N at screen coordinates X,Y.
**ELSE.** An extension of the IF . . . THEN statement unavailable in Applesoft. ELSE allows you further to modify conditionals past the one operation of IF . . . THEN.
**END.** Command to end a program.

**field.** A piece of data in a text file.
**file.** In essence there are two types of Apple files: program files and text files. A program file contains just that; the letters A and I before a catalog name indicate Applesoft and Integer Basic respectively; the letter B indicates machine language. A text file, which can be either sequential or random-access, is a collection of data in a series of so-called fields and is indicated in the catalog by the letter T.
**flag.** A variable for marking the existence of certain conditions within a program. Flags are usually either on or off, but they can be as complicated as you want to make them.
**FLASH.** The command that makes print on the screen blink.
**flow.** The course of a program; a flow chart is a map of this course.
**FOR . . . NEXT.** A FOR . . . NEXT loop allows you to do the same thing over and over again, substituting different values. FOR . . . NEXTs can go either up or down, depending on the STEP value (the STEP default is 1). So you could say FOR X=1 TO 100 STEP 10 or you could say FOR MAMBO=100 TO 25 STEP -25 or simply FOR ZSAZSA=1 TO 10—it's all the same to the computer. There is much discussion of FOR . . . NEXT throughout this book; its complications are interesting and many.
**FRE.** Does one of two things. It either tells you how much space is left for your program, or else it removes all the string garbage your running program has been leaving behind it.
**GET.** An INPUT command that does not necessitate a Return. GET X or GET X$ pauses the program until the user hits any key. Processing then continues, and whatever key was pressed is the value of the variable requested.
**GOSUB.** A branching command to a subroutine—i.e., GO SUBroutine. At the end of the subroutine the command RETURN will send processing to the command immediately following the original GOSUB. *Compare* GOTO, and *see also* POP.
**GOTO.** The branching command from which you never return. A command GOTO (SOMENUMBER) will take you to that line number, and processing will continue from that spot. *Compare* GOSUB.
**GR.** The command to set the video screen to the low-res graphics mode.
**hexadecimal.** A numeric system based on the number 16, with values ranging from 0 to F, 10 to FF, etc. Used because 8-bit processors such as the Apple have 16 fingers, 8 on each hand.
**HGR.** The command to set the video screen to the hi-res graphics mode.
**high-level language.** A computer language that makes fair sense to humans and is easy to use but that needs some translation before the machine can understand it. Basic is a high-level language.
**high-resolution graphics.** Nifty tiny dots by the hundreds, manipulated either by plotting or as bit shapes.
**hi-res.** *See* high-resolution graphics.
**HOME.** A command that clears the screen of all text, placing the cursor at the topmost left corner of the screen.
**housekeeping.** *See* initialization.
**HPLOT.** Hi-res version of PLOT (q.v.).
**HTAB.** The function by which you can set the horizontal position of the cursor, anywhere from 1 to 40. (Sorry, 80-columners; see Chapter 10 for HTABs in the lands of 80.)

**initialization.** The process within a program whereby the program variables are defined and, if necessary, set.
**INPUT.** The command requesting from the user either string or numeric input, as in INPUT X, INPUT "WHAT'S YOUR NAME?";N$, INPUT ZITI$. Processing pauses until information is typed in, followed by a Return. *Compare* GET.
**INT.** This command, followed by a variable in parentheses—e.g., INT(X)—rounds down the value of that variable to an integer. So INT(X) where X = 2.66 would be 2.
**Interpreter.** In Basic, the Interpreter takes each program line as it is processed and translates it into machine-code operations the computer can understand. *Compare* compiler.
**inventory.** The sum of the items that a player puppet is carrying on his person in an adventure.
**INVERSE.** The style of printing on the screen where black letters are overlayed on white, as compared to NORMAL, which is white characters on black.
**kilobyte.** One thousand bytes.
**LEFT$.** Expressed as LEFT$(X$,N), where X$ is some string and N is the number of letters you want counting off from the left.
**LEN.** As in LEN(MY$), a measure the length of a string.
**LIST.** The command to list a program.
**LOAD.** The DOS command whereby programs are read off the disk and placed into the machine memory. Applesoft programs start at machine address $801 (or $800, if you'd prefer to believe that the first byte of every Applesoft program is 00).
**logical exclusion.** In a program, certain conditions exist at any given time. As your programming lines test for more than one of these conditions, once you've eliminated all the possibilities but one, you don't have to test for that one as well. That is, if X can have only a value of either 1 or 2, it is unnecessary to program like this:
40 IF X = 1 THEN 200
50 IF X = 2 THEN 300
If X doesn't equal 1 it *has* to equal 2. When you start combining multiple conditionals on one line, this kind of clearheaded thinking becomes very important.
**loop.** The process by which you keep processing through one or more lines in a program. A limited loop would be a FOR . . . NEXT; and infinite loop would be any form of this:
10 GOTO 10
Our adventure module, long as it is, is a 99 percent infinite loop, the only exits being winning or losing.
**lo-res.** *See* low-resolution graphics.
**low-level language.** A computer language difficult for humans but easy for the computer to understand. Machine language, a series of hexadecimal numbers, is about as low as we can go, using an assembler to get us there.
**low-resolution graphics.** The clunky graphics comprised of blocks, easy to manipulate with plotting commands.
**megabyte.** One million bytes.
**MID$.** Either MID$(X$,4,1) or MID$(X$,9). In the first case we would go to the middle of string X$ and count off one character starting at the fourth

character. In the second case we would start at the ninth character and count off to the end.

**monitor.** The video screen. Note that there is another meaning of monitor referring to hex level operations, which was never mentioned in this book.

**nesting.** Putting small loops inside big loops; e.g.,

```
FOR X = 1 TO 5
FOR Y = 10 TO 20
IF X = Y THEN ? "INCREDIBLE!"
NEXT Y,X
```

You can also nest GOSUBs within GOSUBs.

**NEXT.** *See* FOR . . . NEXT.

**NORMAL.** *See* INVERSE.

**NOT.** A Boolean operator. Evaluates whether a statement is false. Can be used as an equivalent of IF X = O, as in IF NOT X. X = NOT X).

**NOTRACE.** The command to deactivate TRACE (*q.v.*).

**null string ("").** A string of zero length, a string with nothing in it.

**ON.** A command that allows multiple branches depending on the value of a given variable. As in ON X GOTO or ON X GOSUB. The ON command will measure the value of X (or whatever); processing will continue on the Xth line referenced after the GOTO or GOSUB. If there is no matching Xth line, processing will simply continue at the next command following the ON statement.

**ONERR.** As in ONERR GOTO [SOMEWHERE]. A command to handle errors. When this command is placed early in a program, the computer will remember the instruction and act appropriately when the time comes, which usually means branching to an error-trapping subroutine.

**OPEN.** The command to find and commence reading or writing a text file.

**OR.** A Boolean operator in which if one condition on either side of the OR is true, regardless of whether the other side is false, the entire statement is regarded as true. *Compare* AND and NOT.

**parse.** In an adventure conversation segment, to parse means to break down the player's input into its component parts.

**PEEK.** A command that asks the computer to tell you the value contained in a given memory address. PRINT PEEK(103) would cause the computer to tell you the number, or byte, at address 103.

**phase.** A series of complications (*q.v.*) within an adventure.

**PLOT.** Lo-res graphics command that draws a box on the screen at the indicated location. In hi-res, using HPLOT, you can draw either points or lines from point to point.

**pointer.** A value in memory that the computer uses to keep track of FOR . . . NEXTs, GOSUB . . . RETURNs, etc.

**POKE.** The command whereby you can put in specific values at specific addresses in memory. POKE 49168, 0, for example, would place the byte 0 at address 49168.

**POP.** The command that deactivates the most recent GOSUB, magically removing our need for a RETURN.

**PRINT.** PRINT statements transmit information wherever directed, either to the screen, a printer, the disk, or any other output device you designate. More importantly, the symbol ? on the keyboard will translate into the word PRINT, so instead of always typing in five letters you can hit one symbol.

**prompt.** A prompt on the screen is that bit of business that tells the user what to do and where to do it. The blinking cursor following INPUT statements is the usual form of this.
**puppet.** The "character" that the player moves through the game universe in an adventure.
**RAM (Random Access Memory).** The free memory in your machine available to you for programming. Some parts of RAM are already taken, in fact, by the text and graphics pages, DOS, program pointers, etc. *Compare* ROM.
**READ.** As a DOS command, the collector of data from a text file. As a Basic-language command, the collector of data from DATA lines.
**REM.** The command that allows you to talk to yourself within a program. REM statements are ignored by the computer, so anything you write in after you type REM is entirely between you and yourself. REMs are useful for annotating complicated ideas, indicating starting points of subroutines, and slowing down programs. Experience alone will determine how much of a REMer you're cut out to be.
**RESTORE.** After you READ the data in a DATA line, that information is no longer available to you unless a RESTORE command says otherwise. RESTORE places the pointer back at the first piece of data.
**RESUME.** The command that does exactly that, internally continuing a program from where we left off.
**RETURN.** At the end of a GOSUB, a RETURN sends processing to the command immediately following the original command that branched off to the subroutine in the first place.
**RIGHT$.** Expressed as RIGHT$(X$,N), where X$ is some string and N is the number of letters you want counting off from the right.
**RND.** As in RND(X). The Apple has the ability to generate random numbers with this command, but unfortunately they are always the same random numbers. *See* Chapter 9 for the lowdown.
**ROM (Read Only Memory).** Memory space unavailable to you as programmer. ROM contains the systems programs that make your computer work.
**room.** In an adventure, any unique area is called a room. The variable R is used to denote which room is which.
**ROT.** As in ROTate. This command allows you to turn a shape on its axis. The amount of turning depends on the SCALE (*q.v.*) of the shape.
**routine.** In this book a routine means any chunk of a program that performs the function we happen to be talking about at the time.
**RUN.** The command to begin running a program.
**SAVE.** A DOS command by which programs are created (or saved) on a disk.
**SCALE.** Hi-res command that allows you to enlarge a shape. Default SCALE is 1 (same size), and you can't get any smaller than that.
**scroll.** Verb, intransitive. A term used to define how the body of text (or graphics) moves on the screen. Whenever a new line of text is added at the bottom of the already-filled screen, all the rest of the text moves up a line, while the top line disappears entirely. That is what we mean by scrolling. In some programs, text (or graphics) can also scroll left and right, a very advanced technique.
**semicolon (;).** The symbol that suppresses carriage returns at the end of PRINT statements. Put another way, the symbol that concatenates PRINT lines.

**shape table.** A collection of bytes that translates into a shape on the hi-res screen.
**6502.** The type of microprocessor inside your Apple II.
**soft switch.** An imaginary switch, the accessing of which changes the screen mode from text to graphics or vice versa. See the manuals for more.
**spaghetti logic.** The use of innumerable GOTOs in a program to the extent that the logic is impossible to follow.
**SPC.** As in SPC(N), counts off N number of spaces from the present location of the cursor and then prints your next message.
**SPEED.** Literally, the speed with which characters are fed to the screen. Values range from 0 to 255 (doesn't everything?); 255 is the speedy default.
**statement.** A complete instruction to the computer within a program.
PRINT is a command.
10 PRINT "IT'S A BEAUTIFUL DAY IN THE NEIGHBORHOOD"
is a statement.
**STEP.** *See* FOR . . . NEXT.
**STR$.** As in STR$(X); takes the number X and turns it into a string by putting quotation marks around it.
**strategy game.** A game of wits for two or more players. In computer strategy games the computer takes on the role of at least one of the players.
**string.** In essence, as far as the computer is concerned, a string is any batch of characters between two sets of quotation marks. "MUGWUMP" would be a string containing the characters which are the letters that make up the word mugwump; "123" would also be a string, as would "123ABC". I assume that the derivation of the term is from a bunch of characters being strung together.
**structured programming.** A style of programming where all the work of the processing is blocked out clearly into discrete task-performing units, making the program both intelligible and efficient. Some languages, such as Pascal, are structured languages; by definition they lead to structured programming. Others, such as Basic, have only the structure we as programmers put into them; the more structured we get, usually the better off we are.
**subroutine.** A self-contained portion of a program. You branch to it from within the program, get your subroutine job done, then branch back whence you came. Designing programs as a series of subroutines is considered high-class aesthetics.
**surge protector.** A piece of hardware that keeps all electrical foibles, short of a complete loss of power, away from your computer.
**syntax.** The "grammar" of a programming statement. Correct syntax allows the line to be processed; incorrect syntax interrupts the processing.
**syntax error.** A mistake in your usage of Basic. Could be anything from a misspelling to misplaced punctuation, and a lot of things in between.
**TAB.** As in TAB(X), begins printing in the Xth column from the left, which is considered column 0.
**TEXT.** Sets the video screen in the text mode.
**text file.** A collection of data fields on a disk. Either sequential or random-access.
**toggle.** To turn on or off an imaginary switch by accessing the appropriate machine memory address.
**token.** For us Applesofters, a term used to define a number or word with a

preset value or a preset meaning within the language. The token for a CTRL-C interrupt is 255, for example. Not a concept worth worrying about.

**TRACE.** The command that tracks each line number of a program by printing it to the screen while the program runs.

**utility.** A programming aid that helps you run or create other programs.

**VAL.** As in VAL(any string or part thereof). Measures the numeric value of same.

**variable.** A symbol for a number that changes when you're not looking. In Applesoft, variables can consist of any combination of numbers and letters (e.g., F3GOGOMC5 is a valid variable) as long as the first character is a letter and not a number. However, the computer operates on only the first two characters (F3 in the example above). Variables are either real number, X, integer, X% (not used in this book), or string, X$, and all can be used as arrays.

**VTAB.** The function whereby you set the vertical position of the cursor from 1 to 24.

**whistles and bells.** The do-nothing parts of the program that provide nice window-dressing details.

**window.** On your video screen, the placement of what you see in front of you as output by the computer. The perimeters of the window are either 40 or 80 characters across by 24 down.

**WRITE.** The command to PRINT data into a text file.

**XDRAW.** Used to erase shapes; the opposite of DRAW (*q.v.*).

# INDEX


Aces, value of, 199
Address, machines, 152
Adventure games, 2, 24, 259
  budgeting of memory, 128–131
  bugging and debugging, *see* Debugging
  changing a game in progress, 122–124
  conversation with computer, *see* Conversation segment
  copy protection, 126–128
  D&D scenario, 124–125
  editing techniques, 112-115
  elementary Apple clock, 117–119
  graphics, 131–133
  introduction in, 23, 35–39;
  debugging, 110-111, 113
  mapping out the area, 12–15
  planning of, 7–8
    of scenarios, 9–12
  planting of movable objects, use of string variables, 16–18, 19–20
  player input, 56, 58-59
  player moves, 117, 118, 119, 123-124
  procedure, 34
    room layout routine, 24–33
  score, 102-103
  simple arrays, 15–16
  situational, 8, 10
  subjects for, 8–9
  subroutines, *see* Subroutines
  verifying of second word, 119–122
  *see also* "Space Derelict"
Adventure module
  format, 23–24
  outline of program, 106–107
"Adventureland," 8
Algorithm
  betting, *see* Betting algorithm
  creation of, 140–141
  speed of, 168
Alphabetical values, comparing of, 48
AND , 77, 82
Apple clock, *see* Clock
Array(s)
  dimensions of, 15
  memory space for, 15
  in poker program: complications, 164–168,
  dimensioning of, 167–168
  simple, for adventure game, 15–16
ASCII characters, 52, 162
Assembler, 148

Backgammon, 258, 259, 260
Bankroll adjusting, 212
Basic language, 148, 150
Basic programs, 2
Bet Factor, 206
Betting
  algorithm, 205-215
  first round, debugging of, 249
  new patterns of, setting of, 231–235
  second round of, 235–238
  variable, 247
Binary number system, 147–148
Blackjack, 256–257
Boolean logic, 77–78
  in poker program, 202, 223
Bridge, 258, 259
Bugging and debugging, *see* Debugging
*Byte*, 189

CALL , 251
Cards
  dealing of, 178
  drawing of
    human draw, 229–231
    on screen, 228–229
  shuffling of, make-believe speed, 168-172
*Casablanca*, game based on, 8–9, 12
Chain utility, 131
Character(s), 15, 31
"Chat" lines, 176–177
Checkers, 259, 260
Chess, 138, 139, 144, 259
Chinese checkers, 260
Chips (poker), quantity of, 180
CHR$(X) , 52 , 163
CLEAR , 55, 243, 244
Clock, 122
  for adventure game, 117–119
  built in, 104
CLOSE , 52
COBOL, 148
Colons, 77
Color monitor, 131–132

Command(s), 23, 24
Communication, mode of, 56
*see also* Conversation segment
Compartmentalization, 150
Compiler, 148–149
Complications, 10
creating of, 7–8
phases, 11
planning of, 10–12
Computer books, categories of, 1
Computer-controlled players, in poker program, 189–191
handling of, 212–213, 215
setting new betting patterns, 238
Computer language, 148
noncompiled, 150
structured, 149
Computer magazines, 188
Concatenation, 46–48, 52, 112
CONTROL G, 39
CONTROL D, 52
Conversation segment
in adventure game, 24; debugging, 111–112; directional commands, 65-71; two-letter words, 72–78; four-letter words, 82–97; fundamentals of, 56–59; mechanics of conversation segment, 63–65; miscellaneous business, 103–105; moving around, 65–72; programming variables, 97–103; routine, 41; three-letter words, 78–82; words and responses, 59–63
in strategy, 246
Copy protection, 126–128
Copyright laws, 9
*Creative Computing*, 189
Cursor, location of, in adventure game, 33

D&D, *see* Dungeons and Dragons
DATA statement, 119–121
Dealer, 181
Debugging, 41
in adventure game, 23, 35-39, 246; editing techniques, 112–115; procedure, 108–112; recap of adventure module, 106–107
in poker program, 200; difficulty in, 246–247
routines, short, 238
in strategy games, 246–250
Decimal numbers, 148, 152
DELETE , 114
Digital computer, operation of, 147
DIM , 55, 111
Disk, 49
DOS, booting of, 127
DOUBLE-TAKE utility, 244
Dungeons and Dragons, scenario, 124–125
Dungeon Masters, program for, 126

Error-handling routine, 127–128
*see also* Debugging, Specific type of error
ESCAPE keys, 235, 236
EXTRA IGNORED message, 114

"Figure hands," 155–156
File(s), 49–53
writing of, 50
*see also* Text files
"Five-Card Draw," 144, 154, 155
array dimensions in, 167–168
betting the hands, 209–215
creating of betting algorithm, 206–209
drawing cards on screen, 228–229
dropping out, 184–187
figuring of straight, 198–204
finishing off game, 238–245
getting draw card, 215–218
handling of human draw, 229–231
housekeeping for, 165–166
introduction, 160–162
matching of pairs, 195–197
megaloops, 192–195
naming of hands, 204–205
ongoing information, 179–181
program flow, 156–157
screen formatting, 173-187
setting new patterns of betting, 232–235
shuffling of cards, 169–172
FOR...NEXT loop, 39, 44, 95–96, 186, 191, 197, 218
Fortran, 148
FRE(O) , 243

Games(s)
defining of, 138
integration into programs, 140–141
level of play, 261
play of, 10, 35
programming, creative process of, 141
saved, 38
replaying of, 52
saving of, *see* Saving of game
selling of, 9
vocabulary, *see* Vocabulary
GOSUB , 40, 78, 124, 151, 206
minimizing of, 152–153
RETURN without, 124
ticks, erasing of, 124
GOTO , 40, 110, 151, 168–169, 206
minimizing of, 152–153
Graphics, 2
for adventure games, 131–133

nonessential, 143
in poker program, 173, 253–255
in strategy game, 261–262

Hard copy, 112
HELLO program, 127
Help routine, 95
Hexadecimal number system, 147–148
HOME command, 161, 175, 243
use in poker program, 178
Housekeeping
in adventure program, 23, 39–42, 111
in poker program, 165–166
HTAB, 33
in poker program, 175–179, 229, 231
placing of, 180

IF...THEN , 113, 229
*InCider*, 189
Infocom adventures, 32
Initialization, 23, 151
*see also* Housekeeping
INPUT , 51–52, 210, 251–252
INSERT , 58, 104
Interpreter, 148, 151–152
Inventory, limit of, 42
Inventory subroutine, 24, 44–45
INVERSE text, 39, 178

Keyboard
communication with microprocessor, 162–163
programming at, 25

Language, *see* Computer language
LEFT$ command, 46, 47, 48, 78, 196
LEN command, 48
Line numbering, 31, 111, 166
LIST , 127
Locked door, concept of, 11–12
Logic evaluations, 78

Machine addresses, 152
Map(s), 34
making of, 9–10, 12–15
Mathematical statements, use of parenthesis for, 77–78
Megaloop, in poker program, 191–195
Megaprograms, 262
Memory, 15
budgeting of, 128–131
ticks in, 124
Microprocessor, keyboard communication with, 162–163
MID$ , 46, 47, 72

"Mission Impossible," 8
Monopoly, 259
Movable objects
dimensioning of variable for, 43
editing techniques, 114
lists of, 34
location for, 17
planting of, use of string variables for, 16–18, 19–20
quantity of, 44; entering of, 110
setting of, 25
weight of, 42

NEXT , 193, 205, 232
NEXT WITHOUT FOR error, 95–96, 218
*Nibble*, 188, 189
Null-string handler, 64
Number(s)
conversion to strings, 47–48
tangible, 15–16
translation into card value, 171
Numeric variables, 16; dimensioning of, 17

Objects, *see* Movable objects
ON statement, 32, 41, 43, 44, 110, 206, 216, 217, 232
ONERR , 127–128
OPEN , 52
OR , 77, 82
Othello, 258

Parenthesis, 171
Boolean operation of, 82
use for mathematical statements, 77–78
Pascal, 149
PEEK command, 163–164
"Pensate", 261
*Personal Computing*, 189
"Phase," 11
Picture(s), integration into basic module, 132
Picture cares, handling of, 225–226
POKE , 2, 31, 32, 164
in poker game, 235–236
for text window setting, 175
Poker game, 142, 143, 153, 188–189, 260
arrays made complicated, 164–168
betting the hands, completing
betting algorithm, 209–215
column format, 174
counter variable, 164
creating betting algorithm, 205–209
debugging draw segment, 249-250
drawing cards on screen, 228–229
features of, 143–144
figuring of hands, 192-195, 198–204, 223, 248-249

finishing off game, 238–241
getting draw card, 215–228
graphics, 253-255
human draw, 229–231
human models, 189-191
introduction in, 160–162
look of game, *see* Screen formatting
matching pairs, 195–197
megaloops, 191–195
naming of hands, 204–205
opponents, 189–191
program flow, 154–158
random number operation, 159–164
screen formatting, 173-187
second round of betting, 235–238
setting new patterns of betting, 231–235
shuffling the cards, make-believe speed, 168–172
sound and light effects, 250–256
*see also* "Five-Card Draw"
Pool game, 138
POP command, 122, 123–124
Pot
adjusting of, 212
size of, 180
PRINT , 39, 52, 175
for writing to a file, 50
Printout, making of, 112
Program(s)
connective of, 131
designing of, 115
integration of games into, 140–141
layout of, 247
renumbering, 108, 108n; running of, 110
rules, 109
running of, 112–113
structured, 116-117, 147-154
types of, 18-19
tightening of, 116
Punctuation, 112

Question marks, 16, 38, 112

Raise Factor, 206
Random-access file, 49
Random factors, 122
Random number
creating of, 159–164
generation, 2
Randomization problems, 2
Randomness, 260
READ...DATA , 119–121
REM , 31, 42, 51, 110, 165
deleting of, 169
RESTORE , 120
RETURN , 52, 232
deleting of, 122
unreturned, 124
RIGHT$ , 46, 47
RND , 163
Room(s), 24
decorations, 76
descriptions, editing techniques, 114
layout, 23–24, 34, 35, 36, 41, 110; analyzing of, 123
secret, 12
storing of, in files, 131
RUN , 81, 127

Sandbagging, 155
Saved game, replaying of, 52
Saved game file, 53
Saving of game, 109, 111, 113–114
routines, 24
file routines, 49–53
Screen
drawing cards on, 228–229
printing on, 175-176
separating of data on, 32
Screen formatting, for poker program, 173–179, 182-184
dropping out, 184–187
40-column format vs 80-column format, 174–175
ongoing information, 179–181
use of VTAB and HTAB, 175–179, 182–184
Scrolling, 31, 32
Semicolon, 38, 112
Sentences, parsing of, 56, 57
Sequential file, 49
reading of, 49–50
Seven-card stud, 154–155
Shape table, 254-256
Soft switches, 132
*Softalk*, 188–189
Sound effects, 143
in strategy games, 250–256
Space, saving of, 131, 165
"Space Derelict," 10
conversation segment, for moving around, 66-72
designing of, 9
directional commands, 72–76, 78
four-letter words, 82–97
game-saving routine, 50–52
losing in, 54–55
manipulating objects in, 18–20
map of, 13, 14f
miscellaneous business, 104–105

outline of program, 106–107
programming variables, 97–103
responses to words, 62–63
scenario, 21–23
scoring, 103
setting of scene, 36–39
simple arrays, 15
three-letter words, 79–82
winning scenario, 53–54
SPC , 161
SPEED , 39
Spelling errors, 113
Statements, *see* Program statements
STR$ , 47–48
Straight five-card draw, 144
Strategy
concept of, 139–140
differentiated from tactics, 139, 206
Strategy games, 2
algorithms, integrating games into programs, 140–141
debugging of, 246–250; *see also* Debugging
developing of, 142–143, 260
features, 137–140
planning the program, structured programming, *see* "Structured programming"
poker, *see* Poker
types of, 256–259
ultimate, 259–262
*see also* "Five-Card Draw"
String(s)
concatenation of, 46–48
conversion to numbers, 47–48
reading of, 46
String variable(s)
commands, 47-48
dimensioning of, 17
input, 39
manipulation, 17
for movable objects, 16–20
Subroutines, in adventure games
concatenation, 46–48
file routines for saving games, 49–53
housekeeping, 39–42
introduction, 35–39
variable variables, 42–45
winning and losing, 53–55
operation of, 150
Synonyms, 58
Syntax errors, 112, 113, 247
System Master disk, 109

TAB , 161
Tactics, 140
differentiated from strategy, 139, 206
TEXT , 54, 55
for poker game, 160–161, 175
Text files, 49
Text window
resetting of, 33, 236
setting of, 31-32; in poker program, 174-175
Tic-tac-toe, 138-139, 153
outline of, 151
Time factor, 122-123
incorporation into adventure game, 117-119
Time-saving, in poker program, 165
"Top-down" design, 150
TRACE , 113
Treasure hunt games, 8, 44
compared to situational adventure, 8
scoring in, 103
winning or losing scenario, 53

User-friendly concept, 102

VAL command, 48
Values, assigning of, to hand, 156
Variable(s)
character of, handling of, 15
finite number of, 50
fluctuation of, 111
memory-eating, 15
for poker program, 164-168
saving of, 50
setting of, 23, 38, 39-42
*see also* Housekeeping
sorting of, 16
values of, 50
variable, 42-45
Variable name, characters of, 202
Video screen, *see* Screen
VisiCalc, 174
Vocabulary
extents and limitations, 56, 58
input, 41
unusual, 58
VTAB, 33
in poker program, 175-179, 186, 231
placing of, 180

Word-processing programs, 109-110
WRITE , 52

YES answer, 39

"Zork," 32

# BASIC ADVENTURE and STRATEGY GAME DESIGN for The Apple®

This, the most complete and comprehensible book of its kind, strips away the mystery from game design and shows beginning-to-intermediate programmers how to create their own sophisticated computer games in Applesoft® Basic.

**For adventure-game designers,** this book has everything you need to program your own complex adventures, including:

- how to create player puzzles
- how to monitor player and room variables
- how to develop an extensive, flexible vocabulary for conversation between the player and the computer
- *plus* a line-by-line breakdown of the original, fully playable adventure game "Space Derelict!"

**For strategy-game designers,** this book covers all the skills you need to design your own challenging games, including:

- how to structure the program as efficiently as possible
- how to transform strategy and tactics into computer algorithms
- how to debug your completed program
- *plus* a line-by-line breakdown of an original, ready-to-run version of five-card draw poker that pits the human player against four computer-controlled opponents

Filled with invaluable, hard-to-find information and professional tips, and complete with glossary and index, *Basic Game Design* is the definitive guide to the subject.

*Cover design by Giorgetta Bell McRee*

ISBN 0-87196-064-8 >$9.95

# BASIC ADVENTURE and STRATEGY GAME DESIGN for The Apple®

MENICK

FACTS ON FILE